A FRAMEWORK FOR INTEROPERABILITY ACROSS HETEROGENEOUS SERVICE DESCRIPTION MODELS

Alexandre Simard

A thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

March 2019

© Alexandre Simard, 2019

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Alexandre Simard**

Entitled: **A Framework for Interoperability Across Heterogeneous Service Description Models**

and submitted in partial fulfillment of the requirements for the degree of

Master of Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Jinqiu Yang

_____ Examiner
Dr. Todd Eavis

_____ Examiner
Dr. Yann-Gaël Guéhéneuc

_____ Supervisor
Dr. Joey Paquet

Approved by _____
Chair of Department or Graduate Program Director

_____ 20 ____ _____
Dr. Amir Asif, Dean
Gina Cody School of Engineering and Computer Science

Abstract

A Framework for Interoperability Across Heterogeneous Service Description Models

Alexandre Simard

Automated Web service processing, composition and execution is a research area that has yielded different service description models that can be implemented using a wide array of standards and technologies. Due to the diversity of their underlying service description models and the specific operational standards and platforms they are using, it has been difficult to fairly compare various research solutions pertaining to Web service processing, composition and execution. All solutions require a Web service description repository, for example, to automatically compose and execute composite services. Different research endeavors have provided original and diverse solutions to these Web service processing problems, many of them being extensions to existing solutions using standard service description models. We propose a highly modular Web service description framework that can allow the user to import, export, search, modify, and enable the composition and execution of services described using different service description models and/or standards. Our service description framework uses a simple and flexible interface to test and compare Web service processing, composition and execution models and algorithms. We evaluate our framework by creating specific instances of our framework to achieve concrete motivation scenarios, which we do successfully, achieving all of our goals.

Acknowledgments

I was fortunate to have many people inspire, motivate, and advise me during my studies. I offer my sincerest thanks to Dr. Joey Paquet for his indispensable supervision and guidance, without which this thesis would not have been possible. I also thank Serguei Mokhov for his valuable guidance, advice, and insight. I thank my labmates and friends, Jyotsana Gupta and Touraj Laleh, for their support, motivation, and advice. Most importantly, I thank my father, Mike Simard, for believing in me, motivating me, and supporting me and my ambitions.

Contents

List of Figures

Chapter 1

Introduction

This chapter gives the reader a brief overview of the research domain, including past research related to the problem we are investigating, as well as existing research solutions related to what we are presenting in this thesis. It also provides a brief overview of our own research. We cover existing problems with related research, and our solution to these problems, including an overview of how we eventually evaluate our work.

1.1 Overview

In this thesis, we build on previous research related to manipulating Web service descriptions. We introduce this research in Section 1.2. With the context of this research, we lay out a roadmap for this thesis in Section 1.3. We then define the problem we wish to solve in Section 1.4. In Section 1.5, we list a series of motivation scenarios which we aim to achieve with our solution, and will be used to determine whether or not our solution solves the previously mentioned problems we pointed out. In Section 1.6, we list the non-functional requirements that we have determined we must meet by analyzing our motivation scenarios. We then list our research goals in Section 1.7. Then, in Section 1.8, we list our research contributions, and finally summarize this chapter in Section 1.9.

1.2 Introduction to the Research Domain

There has been a wide variety of research problems and solutions that involve the processing of Web service description records. For example, different service composition and execution models have been implemented [1–3], many relying on different standard implementation formats (e.g., WSDL, WADL, BPEL, etc.). For this purpose, basic Web service definition record models have been extended to include notions such as context, constraints, quality of service, and policies, yielding extended models [1–4]. Most of these in turn often require different data specifications to be part of the services' descriptions. In order to evaluate all these different but related solutions, individual researchers traditionally had to customize or adapt existing service description datasets to the specific needs of their new problem or solution. This divergence in the required datasets in turn made it more and more difficult to effectively and fairly compare the different solutions using the same experiments and datasets.

In the past, Web service description record datasets have been created for research purposes through various means, such as Web crawling [5–12], and dataset generation [13, 14]. Other datasets have been offered as public datasets for research [15–19].

1.3 Roadmap

Our ultimate goal is to solve the problem outlined in Section 1.4. We present motivation scenarios in Section 1.5, in which we describe scenarios in which current solutions are insufficient. Later, in Section 1.6, we list non-functional requirements which are fulfilled by implementing the motivation scenarios we describe. To give context to our motivation, we outline existing solutions and research on which we base our own work in Chapter 2. We then describe the architecture required to fulfill the previously described motivation scenarios and requirements in Chapter 3. In Chapter 4, we then describe our implementation of this architecture, which we use

to verify that our motivation scenarios and non-functional requirements can indeed be achieved by implementing the architecture which we have designed and implemented. Finally, we describe the tests we designed to verify that our implementation does fulfill the previously mentioned requirements in Chapter 5, along with the results of these tests. Finally, we summarize our research and lay out our final thoughts in Chapter 6.

1.4 Problem Statement

Essentially, there is a lack of a solution that allows users to aggregate Web service descriptions from various sources and formats, and to adapt Web service description datasets to different formats and Web service description models.

In order to test Web service composition algorithms, one has to potentially combine service descriptions from multiple datasets and/or dataset generators. Each of these datasets and generators represent Web service descriptions in specific formats, use a specific Web service description model, and require specific tools to use them effectively. Previous research has produced real-life Web service repositories [15,16] or tools to produce datasets that do not necessarily represent real-life services available on the Web. These include Web crawlers that search the Web for large numbers of Web service descriptions to be used as a repository for Web service composition [5, 7–11, 20–22]. Other solutions are able to generate artificial datasets based on the user's specifications [13]. Often, researchers invent custom solutions as an extension of another solution, as we mentioned in Section 1.2, such as adding notions like quality of service, constraints, context, or policies to the Web service composition problem [1–4]. In such cases, it becomes even more difficult, if not impossible, to find suitable Web service description datasets to properly test such a solution and compare its performance with other existing solutions that might not use such additional information. Additionally, many research endeavors often require the use of novel data models. Our solution aims at providing a flexible and extensible platform to enable researchers to use existing datasets and automatically transform them to suit

the needs of their own particular solutions and/or data models.

1.5 Motivation Scenarios

We imagine the following motivation scenarios that exemplify situations that are happening in the context of research endeavors related to service computing. All theses scenarios, each in their own way, highlight some problems that are not solved by existing solutions. We not only aim at solving these problems individually, but also to provide a global solution to all these problems under the same solution.

The following motivations scenarios evolved over time. Initially, we had a list of motivation scenarios that represented the requirements our research group had while working with varied Web service description models and formats. In particular, our motivation scenarios were inspired from our attempts to compare different incompatible algorithms that required different formats or Web service description models. We then looked at existing research that faced the same problems and included scenarios that reflected the requirements of this existing research as well. During the development of our solution, we also found that additional requirements emerged, for which we included additional motivation scenarios.

1.5.1 Dataset Translation

A researcher has Web service description record data encoded in file format A, which they wish to process with algorithm X, which itself, by design, requires data encoded in a different file format B. This researcher cannot process their data with the desired algorithm X, since file format A and algorithm X are incompatible. By feeding the data encoded in format A into our solution and outputting an equivalent dataset encoded in format B, this researcher can now feed their data into algorithm X by using the equivalent translated dataset.

More specifically, this researcher may have a WSDL dataset, and wish to feed it to an algorithm that expects input encoded in WADL. The WSDL dataset can be parsed with our solution, and outputted to the filesystem encoded in WADL.

Alternatively, this researcher might have Web service definition records that use a Web service definition record model that includes supplemental information S, which they wish to process with algorithm Y, which requires data augmented with supplemental information T, not included in the researcher's data. Again, this researcher cannot process their data with the desired algorithm Y, since algorithm Y requires supplemental information T. The researcher could use our solution to read in their data, strip away supplemental information S, add information T to the data, and output it to a newly generated dataset file. The data would then be compatible with algorithm Y.

More specifically, this researcher may have a Web service definition record dataset that includes service constraint information [23–25], and wish to feed it to an algorithm that requires quality of service information instead, which is a subset of service constraint information. The researcher can feed their dataset into our solution, strip away the constraint information, generate quality of service information, embed this quality of service information into these Web service definition records, and output the result back onto whatever storage they choose, such as a file or database.

This usage scenario can be abstractly summed-up as the following, which becomes our first functional requirement ***FR1: the solution shall enable a user to translate an existing service description dataset currently expressed using a specific data format/model to another data format/model that enables the representation of the same information in a different format/model.*** We test that our solution can attain this functionality in Section 5.3.1.

1.5.2 Dataset Combination

We imagine a scenario in which a researcher has created an algorithm Z. This time, we imagine that this algorithm requires a large amount of data to be properly tested and quantify some of its qualities. This researcher can use our system to import as many datasets as necessary, each one as large as necessary. When the researcher is satisfied that enough data has been accumulated, the data can then be exported into a new

integrated dataset with necessary information added or removed as explained in the previous section. This researcher can then read this data and feed it to algorithm Z for testing purposes, gathering the required empirical data on the performance of this algorithm.

More specifically, a researcher may have created a new service composition algorithm, and wants to test the scalability of their new service composition algorithm. To do this, this researcher plans to combine together many datasets, some of which are in WSDL, the others being in WADL, and store them in a single format.

This usage scenario can be abstractly summed-up as the following, which becomes our second functional requirement ***FR2: the solution shall enable a user to combine two or more existing service description datasets, which can each be expressed using different specific data formats/models, into a consolidated dataset expressed in a single data format/model that enables the representation of the information stored in the original datasets.*** In cases where the data models involved are different, this functional requirement builds upon our first functional requirement (***FR1***) for the translation of the data as it is being consolidated. We test that our solution can attain this functionality in Section 5.3.3.

1.5.3 Time Scalability

We imagine a scenario in which a researcher is dissatisfied with the time performance of a certain module of our solution. Using an unoptimized module may have been satisfactory when processing time was relatively small, but becomes unacceptable as the size of the processed data increases. In order to optimize the processing time, the researcher should need to only replace the inefficient module with one that offers better performance without having to change other parts of the solution. In other words, there should be no inherent limitation in the design of our solution that prevents a researcher from using or creating a module which offers the appropriate level of time scalability for their particular needs.

This usage scenario can be abstractly summed-up as the following, which becomes

our third functional requirement ***FR3: the solution shall enable a user to achieve time scalability by identifying modules that do not scale with regards to processing time, and easily replace them with other modules that meet their current time scalability expectations.*** This is in fact a functional requirement (i.e., being able to replace a faulty component) that comes from the realization that a non-functional requirement (time scalability) is not met. The non-functional requirements will be expressed in Section 1.6. We test this scenario in Section 5.3.2.

1.5.4 Space Scalability

This scenario is similar to the previous one. We imagine a scenario where a researcher needs to store a dataset that is too large to be handled by one module due to space restrictions, such as when the entire dataset, in whole or in part, is required to be in memory to properly write the information to the file system in a certain format. For datasets past a certain size, this is no longer feasible. Instead, a new module that allows the entire dataset to be properly saved while being less restricted by space limitations is required. In such a case, a new module can be designed and implemented that, for example, stores the information in a database, which is not restricted to the amount of information that can be held in memory. In other words, there should be no inherent limitation in the design of our solution that prevents a researcher from using or creating a module which offers the appropriate level of space scalability for their particular needs.

This usage scenario can be abstractly summed-up as the following, which becomes our fourth functional requirement ***FR4: the solution shall enable a user to achieve space scalability by identifying modules that do not scale with regards to the size of the input, and easily replace them with other modules that meet their current space scalability expectations.*** This is in fact a functional requirement (i.e., being able to replace a faulty component) that comes from the realization that a non-functional requirement (time scalability) is not met. The non-functional requirements will be expressed in Section 1.6. We test this

scenario in Section 5.3.4

1.5.5 Inter-operation with Independently Designed Modules

A researcher may have an existing implementation of an algorithm of their own design, based on the fulfillment of the goals of their own research. This implementation can require data in a specific data format/model that may be specific to their own research needs. In such a case, any data used by this algorithm must either already be in the correct format, or be translated to the correct format. We imagine that this researcher has a dataset in format A, and an algorithm implementation that expects data in format B, and wishes to feed the dataset in format A to the algorithm implementation. This researcher would use our solution to take the dataset in format A, read it, output it as a new dataset of format B, and then feed this new dataset to this algorithm implementation.

This usage scenario can be abstractly summed-up as the following, which becomes our fifth functional requirement ***FR5: the solution shall enable a user to easily connect their own independently implemented modules to operate in conjunction with the features provided by the solution, while minimizing the necessity of these modules to be redesigned.*** We test this scenario in Section 5.3.9.

1.5.6 Maintaining Efficiency along with External Inter-operation

We imagine that a researcher wishes their independently-designed modules to inter-operate with our solution, while maintaining total control over their own implemented code, so that the required connection with our solution interferes as little as possible with the functionality of their own design. For example, a researcher might have developed an extremely well-optimized algorithm and may want to test its performance in a way that peripheral tools are not interfering with its operation. This researcher may have existing datasets available which were used by other solutions

which they are aiming to prove that their solution is superior to. In this case, using the same data set for comparison purposes is absolutely necessary. It may then happen that such datasets are incompatible across the two solutions, which requires translation of the datasets in order to replicate the same experiments using different solutions.

In this case, the researcher would still require the flexibility of our solution and need to incorporate existing dataset translation functionality, such as already available in our solution, but yet they would want their own solution's performance to be totally independent from the processing overhead incurred by the mechanisms implemented in our solution. Their solution could, for example leverage a translation step happening before their own processing, but should not have to interact with our solution inside of their own processing.

This usage scenario, which is a variation of ***FR5***, can be abstractly summed-up as the following, which becomes our sixth functional requirement ***FR6: the solution shall enable a user to easily connect their own independently implemented modules to operate in conjunction with the features provided by the solution, while not necessitating internal coupling inside the modules in question.*** We test that we can achieve this in Section 5.3.5.

1.5.7 Locally Composing and Executing Ready-Made Solutions

We imagine that a researcher has some specific processing steps to effectuate, but does does not wish to write their own code to implement them. They may wish to leverage and combine existing solutions in a way that necessitates that they shop for select existing solutions from an existing catalogue, select them, and rely on a solution that would easily enable the selected functionalities to interoperate without user programming involved. Because the researcher may be concerned with making their data public, they may wish for the processing to be done locally and neither the data being processed, nor the computing itself involve networked communication.

This usage scenario can be abstractly summed-up as the following, which becomes our seventh functional requirement ***FR7: the solution shall enable a user to easily select different pre-implemented modules, which are then made to automatically inter-operate to provide a required processing result, while the processing in question is computed locally.*** We test that we can achieve this in Section 5.3.6.

1.5.8 Connecting to Public Ready-Made Solutions and Datasets

This situation is assuming the same functional scenario as exposed in the previous section, but in cases where the researcher in question is willing to share their data or to use publicly available data, and wants to have access to as many publicly-available ready-made solutions as possible. In this case, a publicly available interface to an instance of our solution is provided. This enables the researcher to leverage the processing elements and datasets incorporated in the public instance.

This usage scenario can be abstractly summed-up as the following, which becomes our eighth functional requirement ***FR8: the solution shall enable a user to easily select different pre-implemented modules, which are then made to automatically inter-operate to provide a required processing result, while the processing in question is computed on a remote server.*** We test that we can achieve this in Section 5.3.7.

1.5.9 Combining Development Strategies

We imagine that a researcher might want to combine the functionality mentioned in Section 1.5.6, Section 1.5.7, and Section 1.5.8. They may want to retain complete control over some critical aspects of their implementation, leverage existing functionality for other aspects that are less critical, and also wish to access publicly available data or processing facilities. Thus, our solution must not restrict a researcher from using a private instance of our solution to communicate with a public instance

of our solution to leverage the advantages of both.

This usage scenario can be abstractly summed-up as the following, which becomes our ninth functional requirement ***FR9: the solution shall enable a user to easily select different pre-implemented modules, which are then made to automatically inter-operate to provide a required processing result, while the processing in question is computed both/either locally and/or on a remote server.*** We test that we can achieve this in Section 5.3.8.

We also wish to clarify that no motivation scenarios which we considered were omitted from this list for any reason.

1.6 Non-functional Requirements

For a solution to allow a user to achieve what we have described in the previously mentioned motivation scenarios, we found that our solution must meet the following non-functional requirements.

Interoperability: *The capability of functional elements of a software system to execute their own process in collaboration with other functional elements, and to share data with other functional elements in a manner that requires each element to have little or no knowledge of the unique characteristics of the internal functioning of other functional elements, especially of what data formats they are using for their own local processing.*

Interoperability enables heterogeneous functional elements to collaborate on the solving of a common problem, even though the different functional elements may use different data models to represent the information that is being processed and eventually shared across functional elements. At the heart of interoperability is the notion of the use of a common data format shared by the functional elements, or, in cases where this is not possible, the incorporation of a translation layer between processing elements that do not share a common data model.

Examples of this include the possibility of using an implementation of our solution

to parse multiple datasets of Web service definition records in different, unrelated formats to import them, or being able to store the parsed information in a database or in a chosen file format. In our case, this is particularly important, since most other requirements we list will be fulfilled by leveraging this requirement. We also need our solution to be accessible to as many use cases as possible to make our solution appealing enough to be used by other researchers and developers.

Adaptability: *The ability of software to adapt its functionality or even its structure according to changes in its environment.*

Our solution should enable researchers to adapt our solution to new circumstances as they arise. For example, the solution should allow a researcher to account for a new file format or Web service definition model if required. This quality is modulated by how statically or dynamically the adaptability can take effect. On the one hand, a software system can be designed with adaptability in mind, so that it is possible to integrate new modules that provide adaptation. The easier the integration of new adaptation modules, the more adaptable the design is. With *dynamic* adaptation, a system has the ability to dynamically detect at runtime that adaptation is required, and to dynamically trigger an adaptation mechanism that may change the structure of the system in order to adapt to an emerging situation.

Extensibility: *The ability of a software system to acquire and integrate new components.*

A researcher should be able to extend our solution to add new functionality. For example, the solution should enable a user to extend existing Web service description records with additional information necessary for specific service models and algorithms to operate. Conversely, the solution should enable the stripping off of certain information from the records in a dataset. This allows users to handle different Web service definition record formats.

This quality is modulated with the *ease* by which the modules can be added.

A system can be designed such that it provides a basic model of execution, which is designed to be extended by the addition of new modules that provide extension points, which are then programmed and added to the code base and integrated after recompilation of the system. On the other end of the spectrum, a system can provide to the user a way to *dynamically* add extension points at runtime. This provides the user with the possibility to extend the capacities of the system as the system is being used.

Scalability: *The ability of a software system to remain usable along with a wide range of increasing loads.*

An instance of the solution should be capable of demonstrating both time scalability (i.e., responding in reasonable time with an increasing load) and space scalability (i.e., processing small to large to very large to extremely large amounts of data) when necessary. In other words, our solution should not restrict its modules from allowing a user to store as much data as needed, and process data as efficiently as possible, without inherently being limited by the design of our solution. Our solution would otherwise be too limited to be useful.

Usability: *The ability of a software system to effectively provide the expected functionalities to the user, in a manner that is as intuitive and the least strenuous or problematic as possible.*

The solution should be as easy and intuitive as possible to use. Our solution must be accessible to as many researchers as possible to be useful. To achieve this, we must ensure that using and extending our solution should require as little technical knowledge as possible. Creating an instance of our solution and extending such an instance with additional functionality should require as few steps as possible.

Throughout the thesis, we justify our design and implementation decisions, as well as our evaluation of the solution, in terms of these requirements.

1.7 Goals and Motivations

In the current situation, different solutions are designed for related problems (see Algorithm A, B, C, D in Figure 1). In some cases, datasets are manually translated to compare algorithms. One such example can be found in the research that used the artificial dataset generator `WSC-Gen` [13, 14] to generate one dataset, which was then manually translated to produce another dataset [26] which was used to develop and evaluate a set of algorithms [4]. Currently, such comparisons are rare due to the amount of effort needed to produce them. For example, we imagine that a generator can be used to generate data to be directly input into Algorithm A and Algorithm B. This data can be manually translated into dataset X and Dataset Y. Dataset Y can then be input directly into algorithm C, compatible with the data in dataset Y, but not dataset X. Dataset X can also be input directly into algorithm D, compatible with the data in dataset X, but not the data in dataset Y, respectively. This is a primitive and inefficient way of using a generator to generate data for different algorithms that require different, mutually incompatible data. We illustrate this in Figure 1.

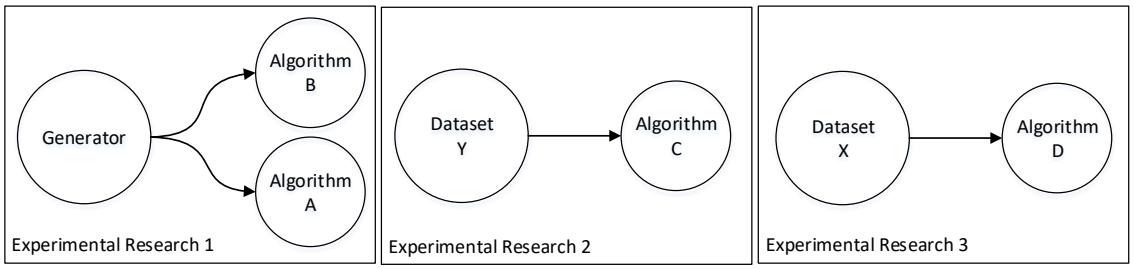

Figure 1: Current situation: Local experimentation

The main focus of this thesis is to provide a flexible solution to the aforementioned problem, the goal being to increase the comparability of various related research results in the field of service computing. We aim to create a solution that is entirely agnostic to the file format, encoding, location, and Web service description model of Web service description records. With our solution, a source of data is first used by a certain researcher, such as a real-life dataset or an artificial dataset generator, and brought into our solution for the purpose of running experiments to evaluate

their solution. The researcher may perform some preprocessing of the data in our solution, such as stripping off information or adding extra fields to adapt Web service descriptions to another Web service description model in order to fine-tune the data for their specific purposes. Once this information is in our solution, it can be published as a public dataset and be accessible by other researchers, which can then be used as-is or possibly adapted to test other algorithm implementations, again using translation layers. Eventually, every dataset relating to Web service description records can be input into the solution and further used to do a comparative analysis of every solution by sharing and translating the datasets that were previously used for an independent evaluation. We illustrate this in Figure 2, where we imagine that each dataset and each algorithm supports a specific, unique, mutually incompatible format.

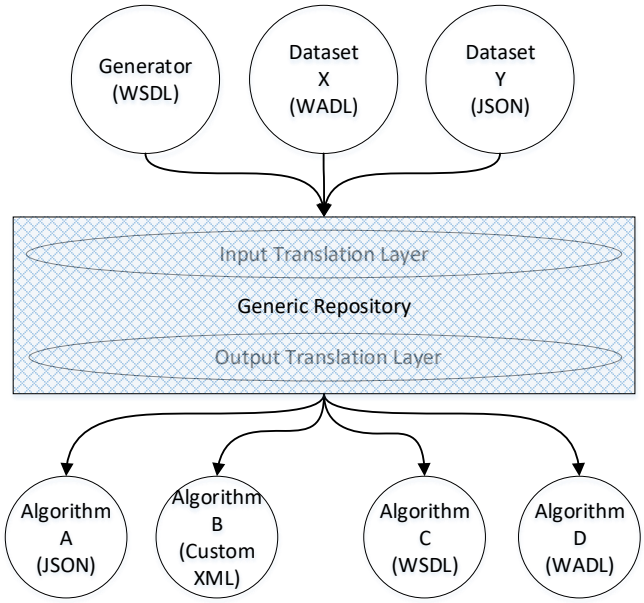

Figure 2: Improved situation: Comparative experimentation

We also allow for algorithms to be developed using our solution. Because our solution allows for any format of Web service description record, algorithms can be developed without being restricted by the format of the Web service description records within any particular dataset. This also allows existing algorithms that were developed in isolation and that each take advantage of separate features within their

respective datasets to be composed together (see Figure 3). This implies that an algorithm, whether pre-existing or in development, can use another algorithm to its advantage, regardless of which Web service description model, file format, encoding, or location it expects. We imagine that our solution could be used to chain together a series of incompatible algorithms, taking the output of one and translating it to the format expected by the output of another. For example, a user could import WSDL created by a generator, WADL obtained from another dataset, and JSON provided from still another dataset, into our solution, and combine all datasets into one. This can be output as WADL for one algorithm, whose output can be sent to our solution, and then output again as BPEL required by another algorithm. The original data created by combining the initial datasets can be used again to chain together two different algorithms in a similar way, the first one requiring WSDL and the second requiring the Lucid programming language to express composite services. We illustrate this in Figure 3.

1.8 Contributions

Our contributions are six-fold.

First, we offer a way for researchers to rely on a platform that enables them to create a service description dataset or repository that can store Web service description records expressed in any Web service definition model, either already existing or of their own novel design.

Second, we offer a way for researchers to develop service processing algorithms that use Web service definition records in their implementation. This implementation can be done independently, or as a composition of pre-existing computational steps chosen by the user.

Third, we offer a way for researchers to compare different Web service definition processing algorithms, even if they rely on different data models or data formats.

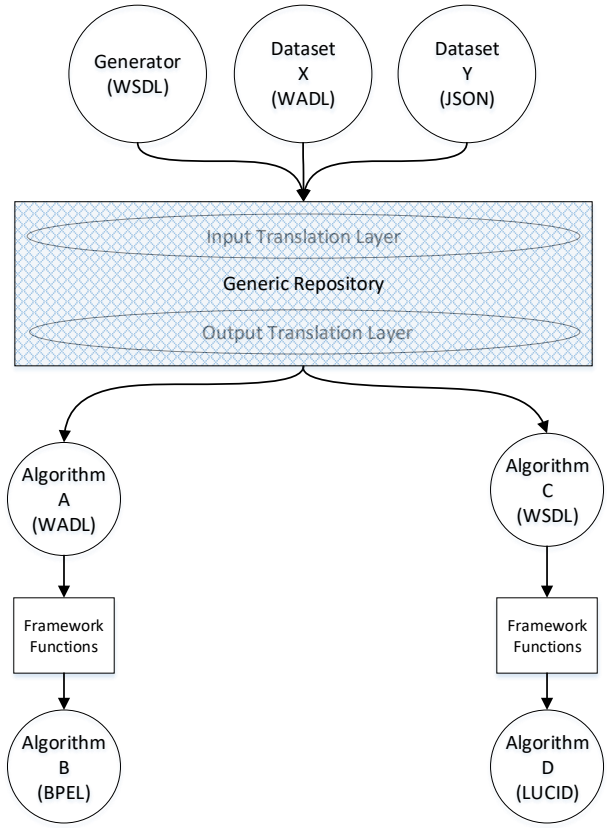

Figure 3: Improved situation: Algorithm composition

Fourth, we offer a way for researchers to use a Web service definition record dataset with an algorithm that is incompatible with its underlying data model.

Fifth, we offer a way for researchers to combine Web service description datasets whose respective service description model or data format are incompatible.

Sixth, we offer a way for researchers to easily produce new Web service description datasets according to their required experimental specifications.

To achieve the above, we have implemented the following features.

First, we offer a way to store Web service definition records using different Web service definition record models. The standard Web service definition record can be extended to include additional information.

Second, we offer a way to read and write Web service definition records in any file format. By simply extending our solution, a new file format, whether a pre-existing standard or a new customized format created by a researcher for experimentation, can be added to our solution.

Third, we offer a way to include the possibility to connect to any database or API that provides Web service definition records. Similarly to reading and writing files containing Web service definition records, one simply needs to extend our solution to account for such a source of Web service definition records.

Fourth, we offer a way to add any system interface to our solution. By simply extending our solution, all the functionality within our solution and any data stored within any instantiation of it can be exposed to an end user in the form of an interface or API. This can include a wide range of functionalities or features.

Fifth, we offer a way to use pre-existing algorithms on Web service description datasets and to develop new algorithms related to Web services and Web service composition. By using our solution, one can gain access to a large number of Web services in many formats, thus making it easier to develop such an

algorithm. Pre-existing algorithms are now no longer restricted to a certain dataset or type of dataset, as other datasets can be translated to a format which a specific algorithm requires.

1.9 Summary

We introduced Web service models and algorithms, ideas which we build on, in Section 1.2. We then laid out a roadmap for this thesis in Section 1.3. The problem we wish to solve was introduced in Section 1.4. To demonstrate that we can solve this problem, we listed a series of motivation scenarios in Section 1.5 that, when achieved, demonstrate that we have solved the problem. From these motivation scenarios, we extracted the requirements that would guide the development of our solution in Section 1.6. All of this lead us to our goals, listed in Section 1.7. We then listed our contributions in Section 1.8.

Chapter 2

Related Work

Our research builds on a large body of previous work that covers many different topics related to Web service processing and related fields. This chapter aims to give an overview of relevant existing research and its relation to our own.

2.1 Overview

In Section 2.2, we list different ways that a solution may offer extensibility, since this is an important part of our requirements listed in Section 1.6. We cover static libraries in Section 2.2.1, dynamic libraries in Section 2.2.2, and frameworks in Section 2.2.3. We then cover the work that has previously been done in the field of Web services in Section 2.3. We cover the formats that have been used to store Web service definition records in Section 2.3.1, the different models that have been conceived for Web service definition records in Section 2.3.2, the datasets that have been created in Section 2.3.3, and the work that has been done on Web service composition in Section 2.3.4.

2.2 Extensibility

We mentioned in Section 1.6 that we require extensibility in our solution. We briefly cover here the ways this has been done in the past and mention their advantages and disadvantages.

2.2.1 Static Libraries

Static software libraries allow one to re-use ready-built modules in one's own solution. This is done by referencing the classes and functions within at compile-time after including the library in their code, including the use of object files [27].

It is important to note that to consider that static libraries offer extensibility, we must consider a very limited, primitive kind of extensibility that requires a large amount of technical knowledge to achieve and is much broader than the definition offered in Section 1.6. In this case, we consider extensibility to be the ability to extend the functionality of pre-existing code in any way. Specifically, in this case, the programmer must know the technical details of the static library to extend its functionality within their own code and make use of this new functionality. Even though the use of static libraries indeed achieves extensibility by allowing the software to extend its existing functionality, we agree that it does significantly diverge from the commonly accepted definition of extensibility, as it severely lacks the crucial aspect of ease of extensibility, or the notion of dynamic extensibility, i.e., software that can be extended by the user at runtime. We have included static libraries as a very primitive example of extensibility for the sake of comparing a wider variety of possible solutions.

Advantages:

- Static libraries allow one to re-use a ready-made solution, avoiding the need to write a solution or some of its parts from scratch.
- Since the entire library is embedded into the executable at compile-time, one does not need to worry about installing any tools or libraries alongside the executable.

Disadvantages:

- Any changes to the library requires re-compiling one's code.
- If two executables use the same library, both executables will load the entire library into memory, effectively placing two copies of it in memory.

- One must be familiar with the details of the library used in order to leverage its functionality.
- One is responsible for controlling the flow of the program and its data.

2.2.2 Dynamic Libraries

Dynamic software libraries allow any executable to reference their internally-defined classes and functions at run-time [28]. One simply needs to reference the library in their code and as long as a copy of it is on their system, the library's functionality will be available to their program. Such examples include DLL files [29].

It is important to note that to consider that dynamic libraries offer extensibility, we must again consider a very limited, primitive version of extensibility that requires a large amount of technical knowledge to achieve and is much broader than the definition offered in Section 1.6. Again, we consider extensibility to be the ability to add functionality to pre-existing code in any way. As with static libraries, a programmer must know the technical details of the dynamic library to extend its functionality within their own code and make use of this new functionality. We use this new definition in this case in the interest of comparing different approaches of extending a solution that would otherwise be difficult to compare, while acknowledging that dynamic libraries are severely limited.

Advantages:

- Changing the library does not always mean recompiling the code, notably if there are no changes to the way one communicates with the library.
- Dynamic libraries allow one to re-use a ready-made solution, avoiding the need to re-write a solution, or even to recompile it.
- Since the library is not embedded into the executable and only referenced, multiple executables can use the same library, while only one instance of the library needs to be loaded in memory to be shared between all of them.

Disadvantages:

- One must be familiar with the details of the library used in order to leverage its functionality.
- One is responsible for controlling the flow of the program and its data.
- Since the library is not embedded into the executable, but is instead only referenced, one must ensure that it is installed on the user's system.

2.2.3 Frameworks

The term *framework* is often misused, so we are explicit in mentioning that we follow the following definition from Wolfgang Pree [30].

Definition 1. ***Software Framework:*** *Software frameworks consist of frozen spots and hot spots. Frozen spots define the overall architecture of a software system, that is to say its basic components and the relationships between them. These remain unchanged (frozen) in any instantiation of the application framework. Hot spots represent those parts where the programmers using the framework add their own code to add the functionality specific to their own project.*

Frameworks inherently provide flexibility and extensibility to a solution. They are particularly popular in Web development and are used in many tools and programs [31–37]. In this case, we no longer consider the limited definition of extensibility that we have previously mentioned, since frameworks allow one to achieve extensibility as defined in Section 1.6. Frameworks can be extended by simply creating a new hot spot, which requires only knowledge of the associated frozen spot. The use of a framework architecture brings the following features to a solution:

Inversion of control: The program flow of software that uses a framework is not controlled by the person extending the software, but by the framework itself [38]. This is useful to us since we want to provide a solution which works in a standard, predictable way for usability.

Extensibility: New hot spots can be created by a developer to add a new implementation to a frozen spot and provide additional functionality, or a variation of existing functionality. This is useful to us as we wish to provide the researchers with the functionality they require to handle their unique data sets and Web service models.

Frozen framework infrastructure: The framework's frozen part should not be modified. Once the frozen spots are established, they cannot be changed, as all the hot spots created are based on a well defined frozen spot. This is useful to us as we require a precise, predictable way to add new functionality to our solution, and the frozen spots provide a generic way for all of the eventually-developed hot spots to interact in a harmonious way.

With this, we can compile the following list of advantages and disadvantages:

Advantages:

- Changing the framework does not always mean recompiling the code of an executable that references it, notably if there are no changes to the way one communicates with the framework.
- Frameworks allow one to re-use a ready-made solution by incapsulating it in a hot spot, avoiding the need to re-write a solution.
- Since the framework can be referenced in an executable the same way a dynamic library is, multiple executables can use the same framework, while only one instance of the framework needs to be loaded in memory to be shared between all of them.
- The framework's frozen infrastructure is responsible for controlling the flow of the program and the data, removing the need of the developer to handle it.

Disadvantages:

- To extend a framework, one must be familiar with its details.

- One cannot easily change the program flow, since it is controlled by the framework's frozen infrastructure. Changing the frozen infrastructure would have the side-effect of potentially invalidating all the previously implemented hot spots.

It is clear that a framework is best suited for our our solution when compared to static libraries and dynamic libraries, as it is the only solution to provide extensibility as defined in Section 1.6.

There are a few ways that one can add a new hot spot to a framework. One such option is to rely on a plugin architecture, which is a common way of extending the functionality of a framework. When a program uses plugins, a user need only download a new component and register it with the host program to add new functionality to the software. For example, plugins have been used to allow users extend the functionality of software such as Eclipse [34, 35], Chrome [36], as well as Atlassian Software [37]. They are useful because they allow a developer to add functionality to a framework without modifying the framework, re-compiling its code [35, 39], or even writing any code at all, as new hot spots can be added by simply downloading a pre-made plugin. This increases the usability of a framework, which means it is a way of improving usability in our solution.

2.3 Web Services

2.3.1 Web Service Definition Record Formats

Any data read by an algorithm must have some kind of structure to be properly interpreted. Web service description records are no exception and must be organized into some kind of structure, format, or encoding to be read into an algorithm that processes Web service definition records. It is common to read Web service definition records from a filesystem. Examples include reading a file to extract the Web service definition records within. More specifically there exist the WSDL [40], WADL [41], BPEL [42], WSLA [23, 24], and PNML [43] formats which we have integrated in the

implementation of various instances of our framework. Many of them are popular formats, and they have all been used online or in previous research as explained in the following sections. Web service definition records have traditionally only been offered in these specific formats, and never in a way that is format-agnostic or with tools that can easily transform a Web service definition record from one format to another according to a user's needs. As a consequence, the repositories, tools, and algorithms related to Web service processing or Web service definition records that have been developed have traditionally been compatible only with its specific format or narrow set of formats [5, 13, 44–47]. For example, SoapUI [45] reads WSDL and WADL files to perform tasks such as Web service simulation and mocking, and cannot import Web service definition records in any other format, as far as we can tell. We read and write the previously mentioned formats to highlight the usefulness of our research and for compatibility with the most widely available data and to show that our solution is compatible with existing research and tools.

2.3.2 Web Service Models

Web service models are at the core of the research in this thesis. They define the data that we wish to help researchers process more easily. We aim to support a wide variety of Web service models to achieve this. To understand them we must first cover some definitions.

We use the following set of definitions [1–3] to guide our work. First, the definition of a *service* provides the abstract structure of the Web service description records that we read and write. In our solution, we consider them to be generic Web service definition records, which contain the bare minimum of information that any Web service definition record handled by our solution must contain.

Definition 2. *A **Service** is defined as a tuple $s = \langle I, O \rangle$ where:*

- ***I** is a set of ontology types representing the input parameters of the service.*
- ***O** is a set of ontology types representing the output parameters of the service.*

As stated before, any alternate definition of a service which we may consider will have to add information to this definition. For example, some extended solutions use the notion of constraints to be applied or verified during service execution. We use the following definition of a *constrained service* [4].

Definition 3. *A **Constrained Service** is defined as a tuple* $s = \langle I, O, C, E, QoS \rangle$ *where:*

- $\boldsymbol{I}$ *is a set of ontology types representing the input parameters of the service.*
- $\boldsymbol{O}$ *is a set of ontology types representing the output parameters of the service.*
- $\boldsymbol{C}$ *is a set of constraint expressions representing limitations on service features.*
- $\boldsymbol{E}$ *is a set of ontology types representing parameters whose values are affected as a result of the execution of the service.*
- $\boldsymbol{QoS}$ *is the set of quality parameters of the service.*

The following accompanying definition, also in [4], is required to clarify the meaning of a *constraint*, and provides information as to what needs to be stored in a constrained service description record:

Definition 4. *A **Constraint** is a Boolean expression. For simplicity, we restrict ourselves to expressions of the form:*
$\langle feature \rangle \langle operator \rangle \langle literalValue \rangle$*, where:*

- $\langle feature \rangle$ *represents a service feature which is an ontology type.*
- $\langle operator \rangle$ *represents operators such as* $=, \neq, <, >, \leq, \geq, \in, \subset, \supset, \subseteq, \supseteq$.
- $\langle literalValue \rangle$ *represents a value or a set of values of the same data type as the expression feature.*

Another related concept is the notion of *composite service*, which can be defined as the following [4], where the composition is stored as a directed graph in a composite service description record.

Definition 5. *A **Composite Service** is defined as a tuple $s = \langle I, O, G \rangle$ where:*

- ***I** is a set of ontology types representing the input parameters of the service.*
- ***O** is a set of ontology types representing the output parameters of the service.*
- ***G** is a directed graph whose nodes are either services or operator nodes that represent one of the following operations: parallel execution, sequential execution, or alternative execution paths.*

Where the service nodes can follow any valid definition of a service.

2.3.3 Web Service Description Datasets

Web service definition record datasets have traditionally been available or gathered in one of the following ways:

There have been several attempts to make a universal registry for Web services, such as UDDI, that were meant to make it easier to find and use Web services, as well as extensions to this to improve its search capabilities [46–48].

There have been Web service description repositories available online as shown in [15–19]. These have been used in research and have produced useful datasets, but have only been able to offer Web service description records in the original format in which they were first found.

Dataset generators have been used before to develop algorithms with success [13, 14]. These allow one to produce a dataset suitable for one's needs when developing a specific algorithm with regards to size and distribution of data, but remain inflexible in some aspects, such as the format of the data produced.

There have been attempts to use Web crawlers to gather Web service definition records into datasets [5–12]. These were still limited and did not offer enough of an improvement to gain significant popularity or widespread use.

As far as we know, there has never been any research into a proper repository of Web service descriptions of various forms that can be queried, that accepts various forms of Web service descriptions, and can translate and export Web service

descriptions in various standard or custom formats. We aim to make these datasets interchangeable and to offer the possibility of combining these datasets, to provide more complete datasets.

2.3.4 Web Service Composition

There has been extensive research into Web service composition and many datasets and dataset generators have been published for and from this research [5, 13–16]. There has also been research on how to find and share services for this purpose [5–12]. Web service composition is a well researched topic that provides us with many algorithms and datasets which we can integrate and compare with our solution. The BPEL format and Petri nets have also been explored as a means of representing Web service composition [49–51], which is why, in the interest of completeness, we chose to include these formats in our solution. There has also been research into using databases to aid in Web service composition [52].

Our research group has researched Web services and Web service composition [1–3]. As such, we base much of the work in this paper on previous research that our research group has produced. Of particular interest are definitions previously discussed in Section 2.3.2, on which we base our implementation, as well as exemplify the capacities of our solution. Additionally, as far as we know, there does not exist a way to compose related algorithms in the field of Web service composition to produce a more complete solution. There has been previous research in the field of inter-cloud interoperability [53–55], which shares many similarities.

2.4 Summary

In Section 2.2, covered the ways extensibility could be achieved as described in Section 1.6, along with all their pros and cons. In Section 2.3, we went over the work that has been previously done in the field of Web services, such as the formats that have been used to store Web service definition records in Section 2.3.1, the different models that have been conceived for Web service definition records in Section 2.3.2,

the datasets that use these concepts in Section 2.3.3, and the Web service composition algorithms that use these datasets in Section 2.3.4. In the next chapter, we explain how we build on these concepts to create our own solution.

Chapter 3

Architecture

In Section 1.7, we outlined what needed to be achieved to attain our goals. In order to meet the stated goals, we chose to design a framework that met all the requirements we had listed, which we implemented in Java. This chapter details the architecture we have designed and explains how it is meant to meet the requirements described in Section 1.6.

3.1 Overview

We cover the design of our architecture in this chapter, explaining how a framework allows us to fulfill our requirements in Section 3.2. We explain how the different frozen spots are interconnected in Section 3.3. We give the specific details of our frozen spots in Section 3.4. Our reader frozen spot is discussed in Section 3.4.1, our writer frozen spot is discussed in Section 3.4.2, our Web service definition record frozen spot is discussed in Section 3.4.3, our translator frozen spot is discussed in Section 3.4.4, our system interface frozen spot is discussed in Section 3.4.5, and the entry point of our framework, our framework loader, is discussed in Section 3.5.

3.2 Framework Design vs. Requirements

We chose to implement a framework after considering our options listed in Section 2.2. We found that this approach was the only one that enabled us to achieve all the qualities we listed in Section 1.6. Below we explain how a framework design enables us to meet each of our requirements.

Interoperability: A framework allows one to combine multiple solutions that were not meant to be combined originally. For example, a framework allows us to use an algorithm on an incompatible dataset. In a framework, we can imagine that there is a frozen spot which defines an abstract transformation, which can be extended by a hot spot that defines a concrete transformation, to allow us to transform data from one format or Web service model to another. To obtain data that is compatible with another format, solution, or dataset, from incompatible data, one need only create a new hot spot that extends this previously mentioned frozen spot, and whose associated transformation is to take data in the incompatible data and transform it to be compatible. The same rationale can be extended to include the cases where we may want to have more than one software component interoperate, even if their underlying data models are different. We describe this more in detail in Section 3.4.4.

Adaptability: A framework provides adaptability, since, if we assume the existence of the appropriate frozen spots, it can account for a new file format or database, including custom formats, by simply introducing a new hot spot. For example, if we have a frozen spot that stores Web service definition records using a certain Web service model, then by creating a new hot spot that extends that frozen spot, we can account for a new Web service definition model in our framework, regardless of the format it was parsed from. We describe this more in detail in Section 3.4.3.

Extensibility: A framework architecure provides extensibility by its very nature. By implementing a hot spot, one has extended whatever functionality was in the associated frozen spot. For example, if the basic Web service definition record is implemented as a frozen spot, then by creating a hot spot that extends this frozen

spot, one has extended the basic Web service definition record. We describe this more in detail in Section 3.4.3.

Scalability: If any of the framework's hot spots process an insufficient amount of data or process data too slowly, one can change these hot spots for new, more efficient ones, so long as there are no inherent mathematical or physical limitations to the processing of this data. For example, if the filesystem is insufficient for a certain task due to the large amount of data that must be processed, a database, which can process much more data, can be used instead. This is done by creating a new hot spot that extends, for example, the reader or writer frozen spots and manages a connection to a certain database, such as PostgreSQL, as opposed to connecting to the filesystem to read, for example, a series of WADL files. If a certain database is insufficient to handle a certain amount of data, a different database, such as a distributed database, may be used instead to provide more storage capacity. We describe this more in detail in Section 3.4.1 and Section 3.4.2.

Usability: By leveraging the inversion of control described in Section 2.2.3, we remove much of the need to worry about the program flow or the flow of data, since much of it has been already defined in the logic and organization of the framework's frozen spots infrastructure. Since extending the framework is done through specific, constrained means that are defined within the structure of the framework, adding a new module becomes much easier. If we use plugins as described in Section 2.2.3, then our framework is much easier to extend. One researcher can implement one hot spot and offer it as a downloadable module to be used by another researcher to take advantage of the work of this first researcher. A researcher could even conceivably perform their entire research without writing any code and only using available plugins. We describe this more in detail in Section 3.6.

3.3 Information Flow

In our framework, readers read the input data, which is then possibly passed to a translator for translation. This translator then produces the Web service definition

records that are added to the list of Web service definition records that are then made available through a system interface. These Web service definition records can then be passed to another translator before being passed to a writer, which will produce the desired output. System interfaces control the readers, writers, and translators to produce the desired datasets and data. This is illustrated in Figure 4.

The framework's frozen spots (in dark grey in Figure 4) are defined to provide the common, basic, format-agnostic structure and functionality of the basic Web service description record (WSDR), as well as the corresponding infrastructure to read and write Web service description elements to and from the repository. The dashed arrows represent control from an interface, and the full arrows represent flow of data.

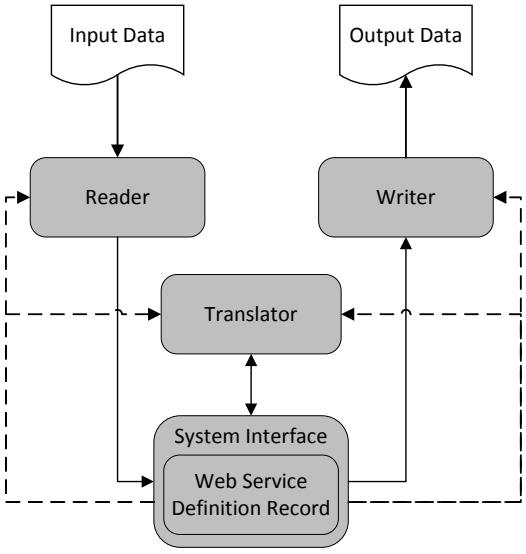

Figure 4: Commands/information flow across framework frozen spots

New sources of Web service descriptions are unrestricted in their format. A modular approach is therefore necessary to allow new formats to be easily accounted for. A central repository is necessary to allow for faster and more convenient searching of services. This allows for a more seamless experience when testing or deploying, for example, Web service composition algorithms.

We have solved the problems mentioned in Section 1.4 by building our solution using an abstract definition of a service description record, presented in

Section 2.3.2, to provide extensibility, as well as a framework approach that leverages polymorphism and enables our solution to read and write records to and from specific, interchangeable formats, to provide adaptability and interoperability. The idea is to have the service description repository built around a skeleton that controls reading and writing, thus providing usability, but where all the processing is delegated to easily interchangeable modules, effectively allowing the repository to allow for a wide range of interfaces and formats to be used, and providing adaptability.

When the framework is first launched, the user must choose which system interface to execute, and when a system interface is selected, the framework controls the execution of this system interface as described in Section 3.4.5. Before a system interface is selected, one may also choose to import a number of plugins, as described in Section 3.6.

3.4 Frozen Spots

The specifics of our framework implementation, as implemented in Java, are described in the class diagrams further in this section.

3.4.1 Readers

In Figure 5, we illustrate the class diagram of our reader frozen spot. We implemented it using the decorator pattern to allow a reader to be extended with any necessary information required for reading service descriptions expressed in a specific data model/format. One achieves this by extending the `ServiceParserDecorator` class, that only requires that one implements a `parse` method that returns an `ArrayList` of `Service` objects, which are described further in Section 3.4.3. Our reader frozen spot has no variables and only has one method, the `parse` method mentioned earlier. The `BasicServiceParser` class, which is our concrete component that describes the basic, default functionality of a reader, implements a very simple `read` method. That is to say, since the basic parser in this case has no idea of how to perform a parse operation, due to its lack of any context or information, it cannot perform a parse

operation, so it simply returns an empty `ArrayList`. Any parsing must be performed by a class that inherits from the `ServiceParserDecorator` class.

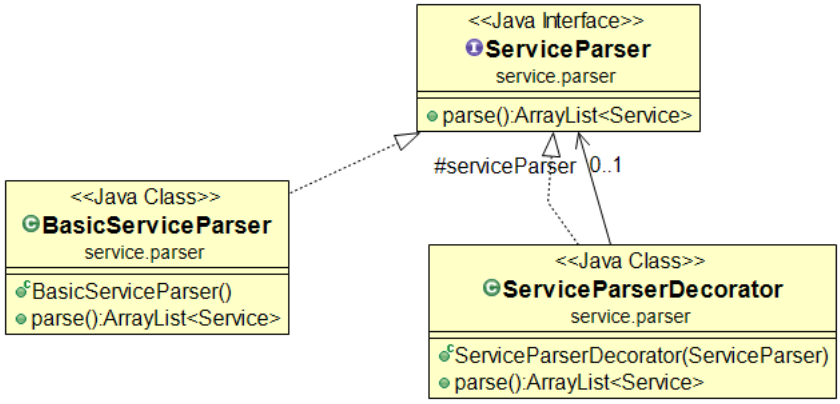

Figure 5: Reader frozen spot

3.4.2 Writers

In Figure 6, we illustrate the class diagram of our writer frozen spot. Mirroring the reader frozen spot design, we implemented it using the decorator pattern to allow a writer to be extended with any necessary information required for writing service descriptions expressed in a specific data model/format. One achieves this by extending the `ServiceWriterDecorator` class, that only requires that one implements a `write` method that has one parameter, an `ArrayList` of `Service` objects. Our writer frozen spot has no variables and only has one method, the `write` method mentioned earlier. The `BasicServiceWriter` class, which is our concrete component that describes the basic, default functionality of a writer, implements a very simple `write` method. That is to say, since the basic writer in this case has no idea of how to perform a write operation, due to its lack of any context or information, it cannot perform a write operation, so it simply performs no operations. Any writing must be performed by a class that inherits from the `ServiceWriterDecorator` class.

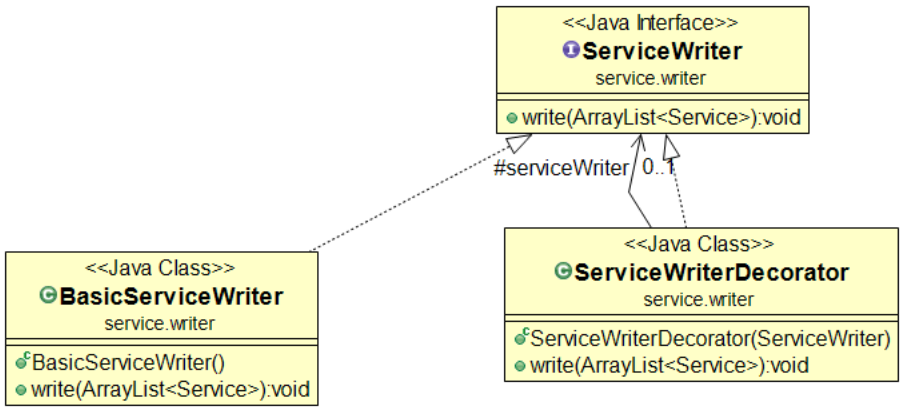

Figure 6: Writer frozen spot

3.4.3 Web Service Definition Records

In Figure 7, we illustrate the class diagram for our Web service definition record frozen spot. We implemented it using the decorator pattern to allow a Web service definition record to be extended with any necessary information required for a certain Web service description model. One achieves this by extending the `ServiceDecorator` class, that requires that one implements the following methods:

`GetName`: This method expects no parameters, and returns a string representing the service's name. We assume that each service has a unique name. In the concrete component, our basic service, this method returns the string representation of the service's name stored in the basic component.

`GetInput`: This method expects no parameters, and returns an `ArrayList` of `String` objects representing the inputs of this particular service. In the concrete component, our basic service, this method returns the `ArrayList` of strings representing this service's inputs stored in the basic component.

`GetOutput`: This method expects no parameters, and returns an `ArrayList` of `String` objects representing the outputs of this particular service. In the concrete component, our basic service, this method returns the `ArrayList` of strings representing this service's outputs stored in the basic component.

`SetInput`: This method expects an `ArrayList` of `String` objects, and sets this

service's inputs to the provided `ArrayList`. In the concrete component, our basic service, this method sets the `ArrayList` of strings representing this service's inputs stored in the basic component to the `ArrayList` that was passed in.

`SetOutput`: This method expects an `ArrayList` of `String` objects, and sets this service's outputs to the provided `ArrayList`. In the concrete component, our basic service, this method sets the `ArrayList` of strings representing this service's outputs stored in the basic component to the `ArrayList` that was passed in.

`GetInnerService`: This method returns the `Service` object decorated by this particular `Service` object. In other words, this method provides a `Service` object stripped of the additional information provided by this `Service` object's particular class. In the concrete component, our basic service, this method simply returns null.

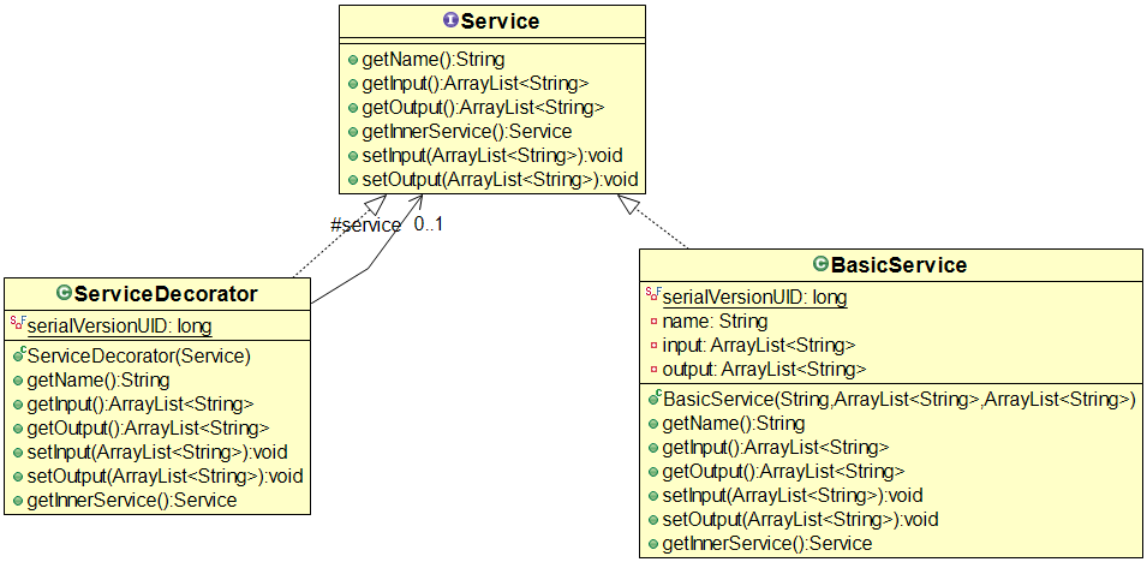

Figure 7: Service frozen spot

3.4.4 Translators

In Figure 8, we illustrate the class diagram of our translator frozen spot. We implemented it as a Java interface to allow a translator the freedom to implement any translation required. One achieves this by implementing the `TranslationLayer` interface, that requires that one implements one method, `translate`, which returns

an `ArrayList` of `Service` objects, and has one parameter, again an `ArrayList` of `Service` objects.

Figure 8: Translator frozen spot

3.4.5 System Interfaces

In Figure 9, we describe the class diagram of our system interface frozen spot. We implemented it as a Java interface to allow a system interface to be implemented in any way a researcher requires for their research. One achieves this by extending the `ServiceDecorator` class, that requires that one implements the following methods:

`open`: This method is responsible for obtaining and initializing all resources used by the system interface. This method does not expect any parameters and does not return any value.

`execute`: This method is responsible for performing the main tasks of the system interface, including controlling input and output, and coordinating the other modules in our framework. This method must ensure its own proper termination. In other words, it must control when the system interface is to be closed. This method does not expect any parameters and does not return any value.

`close`: This method is responsible for freeing any resources used by the interface during execution and performing any tasks necessary for closing this interface. This method does not expect any parameters and does not return any value.

These three methods will be executed in the order shown above by the framework we have produced once a system interface is selected. The other methods accessible to the system interface are already defined and implemented by the parent class. They are as follows:

`readers:` Provides a list of all the readers available at the moment by returning an `ArrayList` of `String` objects. No parameters are expected.

`writers:` Provides a list of all the writers available at the moment by returning `ArrayList` of `String` objects. No parameters are expected.

`translationLayers:` Provides a list of all the translators available at the moment by returning `ArrayList` of `String` objects. No parameters are expected.

`interfaces:` Provides a list of all the system interfaces available at the moment by returning `ArrayList` of `String` objects. No parameters are expected.

`webserviceDatasetFiles:` Provides a list of all the files available at the current working directory on the filesystem by returning `ArrayList` of `String` objects. No parameters are expected.

`saveToDatabase:` Saves all services currently in memory to the database. No values are returned and no parameters are expected.

`loadFromDatabase:` Loads all the services in the database to memory. No values are returned and no parameters are expected.

`readFile:` Reads a file from the filesystem to extract the Web service definition records within, adding them to the `ArrayList` of `Service` objects already contained in this system interface. No values are returned, and the parameters expected are one string representing the reader to be used, and another string representing the location of file to be read from.

`writeFile:` Writes a file to the filesystem to save the Web service description records currently in memory. No values are returned, and the parameters are one string representing the writer to be used, and another string representing the location of the file to be written to.

`searchForService:` Looks through the services in memory and searches for one with a matching name. There are two versions of this method, one which returns a string representing the service that was found, but formatted according to a particular

file format, such as JSON. In this case, the method expects two parameters, the first being a string representing the name of the service to be searched, and the other being a string representing the writer to be used to generate the returned string. The second version of this method simply accepts one string representing the service to be found, and returns the corresponding `Service` object.

`loadReaderPlugin:` Adds a reader plugin to the list of available reader plugins. No values are returned, and only one parameter is expected, namely the name of the reader to be added as a plugin.

`loadWriterPlugin:` Adds a writer plugin to the list of available writer plugins. No values are returned, and only one parameter is expected, namely the name of the writer to be added as a plugin.

`loadTranslationLayerPlugin:` Adds a translator to the list of available translator plugins. No values are returned, and only one parameter is expected, namely the name of the translator to be added as a plugin.

`loadInterfacePlugin:` Adds a system interface to the list of available system interface plugins. No values are returned, and only one parameter is expected, namely the name of the system interface to be added as a plugin.

`translateServices:` Sends services to a translator to be translated. Two versions of this method exist, the first expects as parameters a string representing the translator to be used, and an `ArrayList` of `Service` objects to send to this translator. The other version of this method only expects a string representing the translator to be used as a parameter, and all services currently in the system interface are sent to the translator. In both cases, an `ArrayList` of `Service` objects, namely, the results of translation, is returned.

`getServices:` Provides all the services currently stored in memory in our framework. No parameters are expected, and an `ArrayList` of services is returned.

Note that all the above methods can be redefined in a new system interface, the functionality is not limited to that which is defined in the parent class, and new

functionality can be implemented in a new system interface.

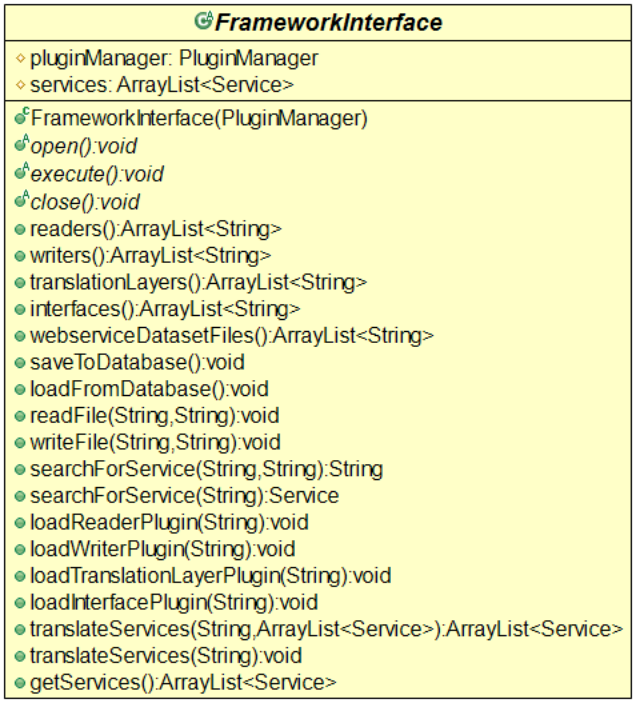

Figure 9: System interface frozen spot

3.5 Framework Loader

The framework loader is a module in our framework that is the main entry point for the program flow. It is responsible for importing plugins when there are no available hot spots, and selecting which system interface to run among those that are available.

When the bare framework is executed, the framework loader will appear as a command line interface that offers the user the choice of either selecting a system interface to execute, or importing a plugin.

If the user chooses to import a plugin at this point, the plugin manager will be called to import the specified plugin. This is described in greater detail in Section 3.6. We offer this functionality since there is a chance that a user might want to obtain a version of our framework that has no hot spots included, so they can have a version that is as small as possible, and allows them to customize our framework with their own hotspots as they please.

When the option of executing a system interface is selected, the user is first presented with a list of available system interfaces obtained from the plugin manager. The framework loader, when a system interface is selected, will call three methods from the system interface as described in Section 3.4.5, defined in all system interfaces. These methods are meant to define a precise, predictable program flow used by all system interfaces, and are the following:

`open:` Responsible for initializing all resources required by the system interface in question.

`execute:` Responsible for handling the input, output, and any interaction with the framework. This method is also responsible for calling the methods defined in the parent class of the system interface.

`close:` Responsible for freeing all resources used by the system interface.

3.6 Plugin Manager

The architecture of the plugin manager we have designed is illustrated in Figure 10. It allows one to import a plugin, store the plugins that have been imported, and obtain an instance of the hot spot defined in any of the imported plugins. The plugin manager is responsible for importing plugins, storing them in a plugin listing, and providing instances of the hot spots within to be used in a system interface.

Practically, our plugin manager is rudimentary. To add a hot spot through it, the Java class that implements this hot spot must first be accessible to the framework. There are two ways to do this, the first being including the Java file with the Java code directly in our framework, the other being including a JAR file containing the class that implements a particular hot spot in our framework. Once this is done, the plugin manager requires the hot spot's class name, including the namespace, to import the associated hot spot. The plugin manager stores these hot spots by storing a list of these class names in a simple text file. The plugin manager can also supply an instance of one of the classes stored in the previously mentioned text file. This

is achieved through Java self-reflection, which provides an instance of an available class at run-time when provided a string of the class' fully-qualified name. Both the framework loader described in Section 3.5 and the system interfaces described in Section 3.4.5 can access the plugin interface. Note that as before, dashed arrows represent control, but newly introduced dotted arrows now represent the flow of a plugin as it goes from a class to an instance with the help of the plugin manager.

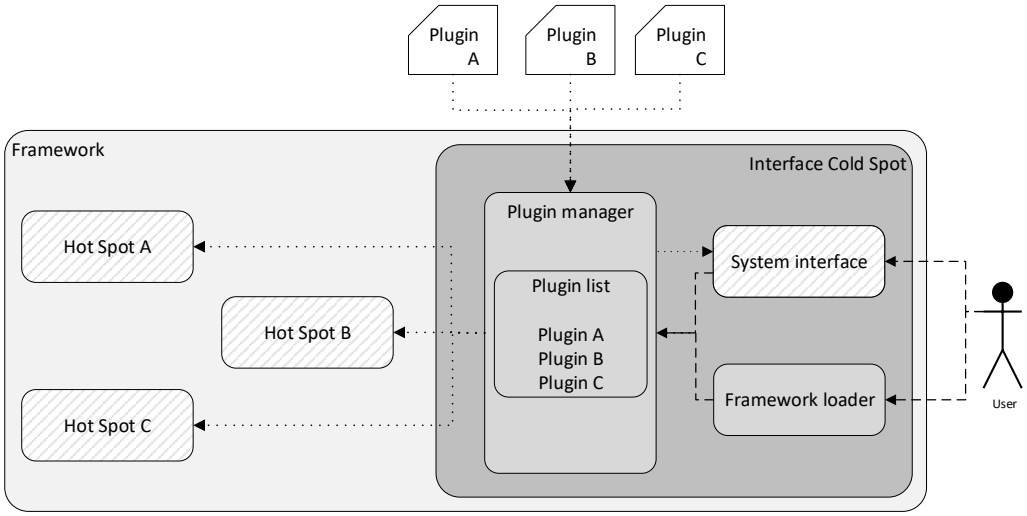

Figure 10: Framework architectural design of the repository: Plugins

3.7 Summary

In this chapter, we introduced the design of our framework. We explained how a framework allowed us to fulfill our non-functional requirements in Section 3.2. We explained how the different frozen spots are interconnected in Section 3.3. We then gave the specific details of our frozen spots in Section 3.4. Our reader frozen spot was discussed in Section 3.4.1, our writer frozen spot was discussed in Section 3.4.2, our Web service definition record frozen spot was discussed in Section 3.4.3, our translator frozen spot was discussed in Section 3.4.4, our system interface frozen spot was discussed in Section 3.4.5, and our framework loader was discussed in Section 3.5. In the next chapter, we describe in detail how the framework can be instantiated as to achieve the motivation scenarios described in Section 1.5.

Chapter 4

Framework Instantiation

In this chapter, we describe in detail how to create an instance of the framework we described in Section 3.

4.1 Overview

This chapter covers the framework instance we created to fulfill our motivation scenarios, as mentioned in Section 1.5. We explain how all of our hot spots are interconnected in Section 4.2. We describe how our system interfaces connect to the rest of our framework in Section 4.2.1, how our readers connect to the rest of our framework in Section 4.2.2, how our writers connect to the rest of our framework in Section 4.2.3, our Web service definition records in Section 4.2.4, and our translators in Section 4.2.5. The specific details on the hot spots we have created are discussed in Section 4.3.

4.2 Information Flow

Our framework instance is illustrated within Figure 11. As described in Section 3.3, our frozen spots (in dark grey in Figure 11), provide the basic structure of our framework that can be extended through framework instantiation. The framework hotspots (with a linear hatching background pattern in Figure 11) exploit and

showcase the following:

1. The expansion of the common basic Web service description records required for our requirement of extensibility.

2. The reading and writing of specific data formats, e.g., WSDL, WADL, WSLA, BPEL, or any custom-defined data formats. This is required to meet our requirement of adaptability and interoperability.

3. Translators that can perform a wide variety of transformations on a list of Web service description records. This also helps us achieve extensibility and usability, as it allows one to use different Web service models in conjunction more easily.

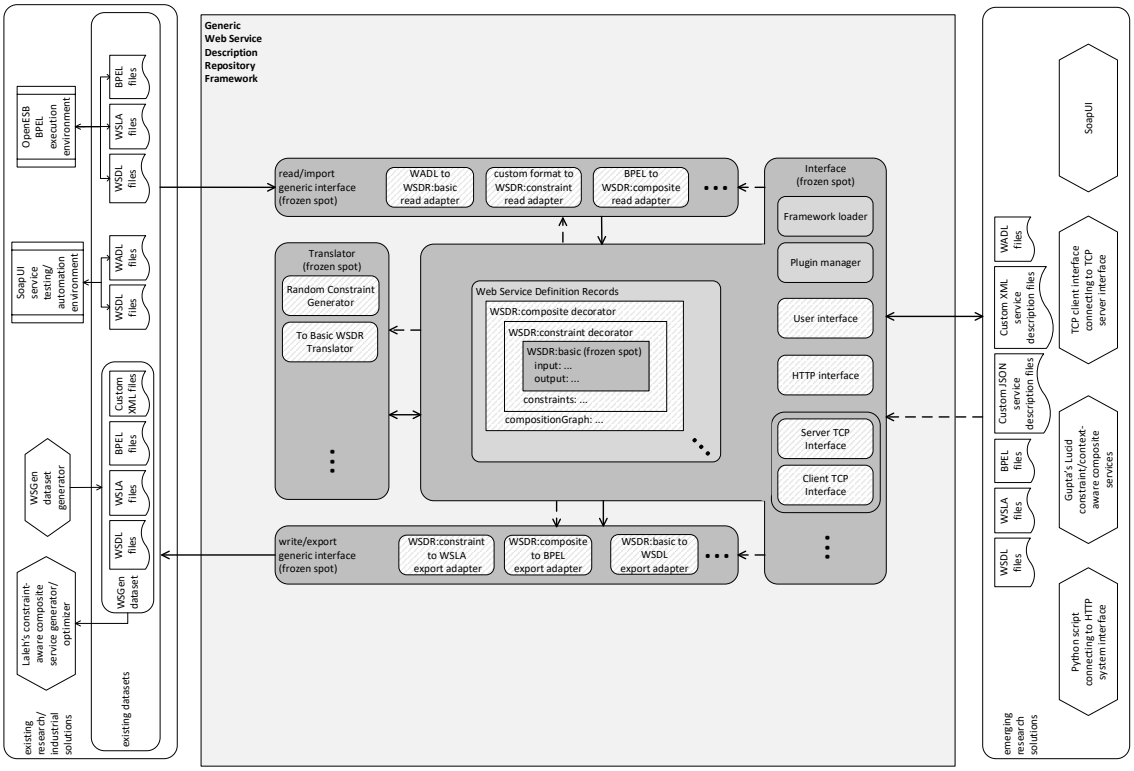

Figure 11: Framework: Example instance

Note that within Figure 11, solid arrows represent the flow of information, and dashed arrows represent the flow of control.

We illustrate the existing solutions and how they interact with data, as well as the details of our own solution and the details of how our own solution interacts with data. This is done to compare our solution to existing solutions while demonstrating the functionality of our solution.

Within existing solutions, a WADL or WSDL dataset can be read by SoapUI [45] to mock the services described within such a dataset. A dataset comprising of WSDL, WSLA and BPEL can be read by OpenESB [56] for processing. The WSC-gen Web service generator [14] produces datasets comprising of WSDL, WSLA, BPEL and XML files. Laleh has created algorithms that take the files produced by the WSC-gen Web service generator to process them [4].

Our framework allows emerging solutions to access data. For example, we can produce and parse datasets in WSDL, WADL, BPEL, WSLA, and our custom JSON and XML formats. Notably, another student in our research group has leveraged and extended the functionality of our framework to produce algorithms related to service composition [57]. We have created two programs which communicate with our framework using various system interfaces, the Python script mentioned in Section 5.3.7 that connects to our HTTP system interface, and our TCP client interface mentioned in Section 4.2.1 which connects to our TCP server interface through a custom TCP protocol. SoapUI [45] has also been connected to our framework through indirect means as described in Section 5.2.6.

The main point of entry of our framework, and the center of control, from the perspective of an external tool, new solution, or algorithm, is our system interfaces. They can receive and produce data, as well as receive commands to read, write, or manipulate data. Within a system interface, there exists a listing of Web service definition records, whose records can be decorated with additional information, such as constraints for constrained Web service definition records, or service graphs for composite Web service description records. Our system interfaces use our reader and writer hot spots to input and output data to and from our framework for processing. Our system interfaces also control our translators to manipulate the data within our framework.

4.2.1 System Interfaces

Our framework includes a framework loader implemented as a frozen spot that allows a user to choose which system interface they may want to use to communicate with their framework instance. This provides interoperability and usability, since this allows us to offer a variety of ways of interfacing with their framework instance, as well as ease for the user to choose what interfaces they wish to activate in order to interact with their framework instance. This functionality is illustrated in Figure 12.

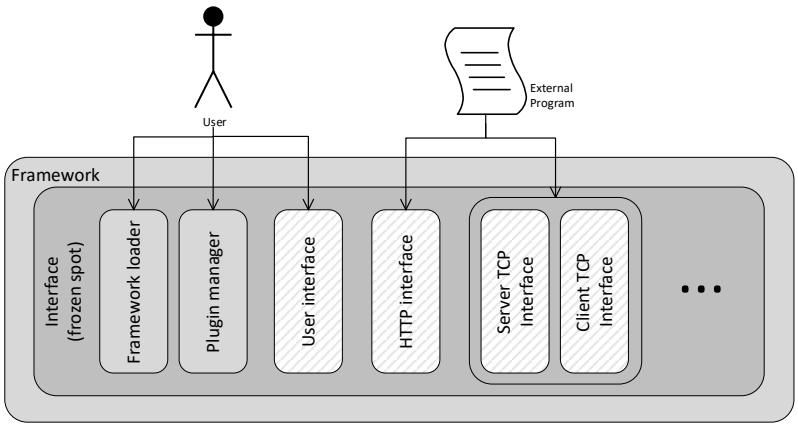

Figure 12: Framework instance: Interface interactions with external entities

As described in Section 2.2.3, frameworks are responsible for controlling the program flow. As described in Section 3.5, our framework loader allows a user to select which system interface to execute from available system interfaces, as well as import new plugins, which may include other new interfaces. A system interface then has access to all the other hot spots available in our framework, such as readers, writers, translators, and Web service definition records. Notably, we have a command line interface that a user may use to interact with our framework. We have also implemented an HTTP interface and an interface that implements a custom protocol over TCP, both of which can be accessed by an external program over the internet. System interfaces can even access the plugin manager to import new plugins or obtain hot spots defined in the imported plugins. This functionality is illustrated in Figure 13. As before, dashed arrows represent control, and the dotted arrows

represent the flow of a hot spot from a class in Jar library, to the plugin manager, to an instance.

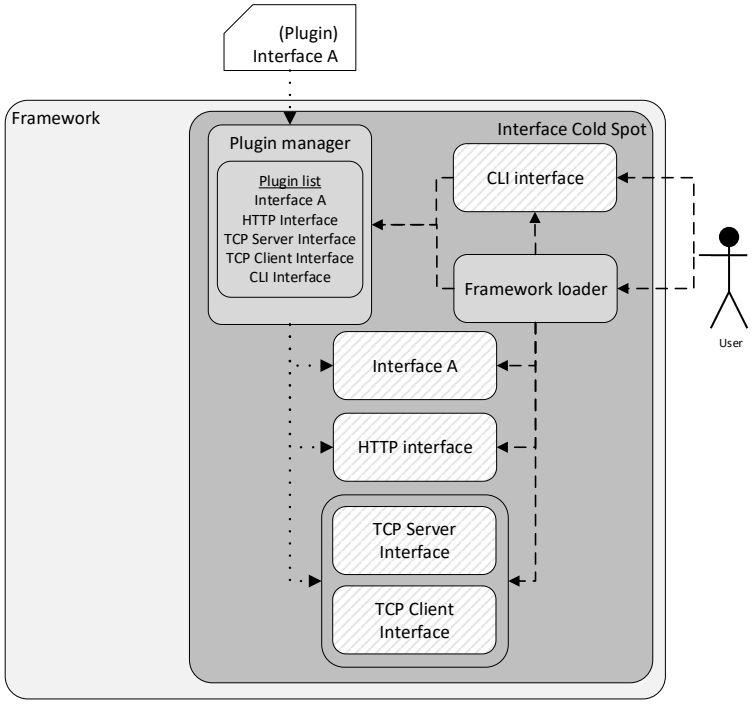

Figure 13: Interface activation in a framework instance

Command Line System Interface

We have implemented a command line interface hot spot that offers the user various options for interacting with our framework instance, such as controlling readers, writers, and translators, as well as importing plugins that extend any of our frozen spots and search for specific Web service definition records. A user uses various menus to select which hot spot to use and provide hot spots with the relevant information necessary for functioning. The options one may choose are the following:

- Retrieving all Web service definition records from a database, adding them to the list of Web services contained within the command line interface hot spot.
- Reading a file from the file system. First a reader is chosen from among the following:

  - An XML reader that reads constrained Web service records in our custom XML format.
  - An JSON reader that reads constrained Web service records in our custom JSON format.
  - A WADL reader that reads basic Web service records.
  - A WSDL reader that reads basic Web service records.
  - A reader that accepts another reader hot spot and a constraint reader. We use this to read a WSDL file and a WSLA file together to read the constrained Web service description records described within.
  - A reader that reads the Java objects in a file containing serialized Java objects that represent Web service description records.
  - The final option allows a user to select from any of the imported reader plugins to use.

- Writing all services contained within the command line interface hot spot to the file system in one of the following formats:
  - Our custom XML format containing constrained Web service records.
  - Our custom JSON format containing constrained Web service records.
  - WADL containing basic Web service records.
  - WSDL containing basic Web service records.
  - Serialized Java objects. This has the advantage of working for any type of Web service definition record.
  - A WSDL file and a WSLA file together, using our separated writer hot spot. This particular hot spot accepts another hot spot and a constraint writer, calling both simultaneously and writing the basic Web service definition records using the provided hot spot, and the constraints using the constraint writer.

  - The final option allows a user to select from any of the imported writer plugins to use.

- Translating all Web service definition records contained within the Web services definition records in this interface. A user is asked to choose which translator hot spot to use from the list of translator plugins that have been imported. The Web service definition records in the system interface are replaced by the ones produced by the translator.

- Searching for a Web service definition record. The user is prompted for a service name, and the names of all the services whose name matches the provided service name are displayed in the interface.

- Saving all Web service definition records to a database. All the Web service definition records currently in the system interface are written to the database.

- Importing a plugin. The user is prompted for the name of the plugin and its type, which are then sent to the plugin manager to install this plugin as a hot spot in the framework.

- Saving all plugins. The plugin manager is instructed to write its plugin to the plugin listing on the file system.

- Reloading all plugins. The plugin manager is instructed to read its plugin listing to determine which plugins are available.

HTTP System Interface

We have implemented an HTTP interface hot spot that starts an HTTP server that waits for the following HTTP requests:

- A **`Post`** request that contains a file with Web service definition records. In the request, the format of the file must be specified so that the proper reader can be used to parse the contents of the file. The formats that one can use are the

following: WADL, WSDL, our custom JSON and XML formats, and serialized Java objects.

- A **`Get`** request that specifies a file format among the following: WADL, WSDL, our custom JSON and XML formats, and serialized Java objects. The corresponding writer will be selected to produce output containing all the Web service definition records currently stored in the database. This output is then sent as the response to the initial request.

TCP System Interface

As mentioned in Section 1.5.9, we need a way to allow a local instance of our framework to communicate with a public instance of our framework. To do this, we have also created a pair of system interface hot spots that communicate using a custom protocol built on top of TCP. Namely, our server and client TCP interfaces. The client interface provides the same functionality as the command line interface mentioned earlier, but also provides the option of communicating with our server interface. The sole purpose of our server interface is to communicate with this client interface.

As mentioned previously, we use TCP to connect the client interface to our server interface. TCP merely allows information to be sent over a network, but, at the application level, does not govern how the information is interpreted, how messages are formed and formatted, or how to appropriately respond to a message. We therefore describe the protocol we have developed for completeness, illustrated in Figure 14.

After a client connects to the server, communication is always initiated by the client, with the server responding as appropriate after each message received from a client. A server always expects to receive a serialized `Command` object. This object contains an integer code representing the type of the command to execute on the server, as well as a list of strings representing the arguments to this command. When the server receives an instance of a `Command`, it then interprets it, executes it, and sends a response back to the client. A response from the server can be of three types:

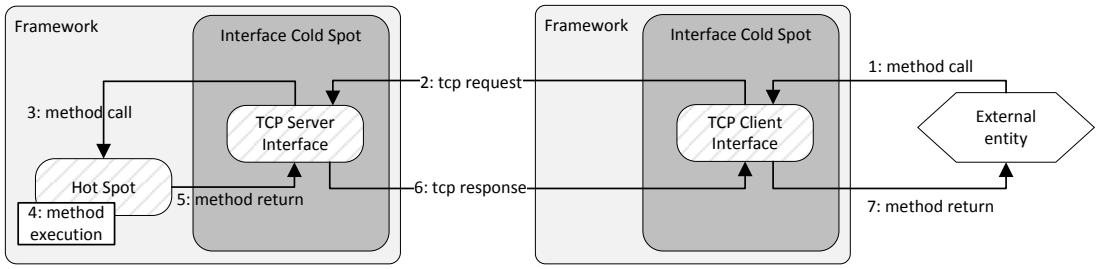

Figure 14: Client-server inter-framework communication using TCP

- A string representing success message in the case of a command that produces no output.
- A list of `Service` objects representing the results of the execution.
- A list of strings representing the results of the execution in all other cases.

Below is the list of commands accepted by the server, along with their command codes:

`Code 1` Get all writers currently available to the server. Requires no arguments. The response is a list of strings.

`Code 2` Get all readers currently available to the server. Requires no arguments. The response is a list of strings.

`Code 3` Get all translators currently available to the server. Requires no arguments. The response is a list of strings.

`Code 4` Import Services from database and store them in memory. Requires no arguments. The response is a success message.

`Code 5` Get list of files available to the server currently in the current working directory. Requires no arguments. The response is a list of strings.

`Code 6` Import Services from file. Requires two arguments, the reader and location of the file to be read from. The services read are stored in local memory. The response is a success message.

`Code 7` Export Services to filesystem. Requires two arguments, the writer and location of the file to be written to. The response is a success message.

`Code 8` Translate Services using a translator. Requires one argument, the name of the translator to be used. The response is a success message.

`Code 9` Search Services in memory. Requires one argument, the name of the service. The response is a list of services containing a single service, if applicable.

`Code 10` Get all services. Requires no argument. Responds with a list of all services currently in the server's memory. The response is a list of services.

`Code 11` Save all services currently in memory to database. Requires no arguments. The response is a success message.

4.2.2 Readers

The readers in our framework are illustrated in more detail with some examples in Figure 15. Our system interface hot spots are responsible for controlling the readers. Some examples of how some system interfaces can control readers to import data include the ability read in a JSON dataset by controlling a JSON reader through either the command line interface or the custom TCP interface. We can similarly read in a dataset written in a custom XML format by using either of our previously mentioned interfaces.

4.2.3 Writers

The writers in our framework are illustrated in more detail with some examples in Figure 16. Our system interface hot spots are responsible for controlling the writers. Some examples of how some system interfaces can control writers to export data include the ability to write a JSON dataset by controlling a JSON writer through either the command line interface or the custom TCP interface. We can similarly write in a dataset written in a custom XML format by using either of our previously mentioned interfaces.

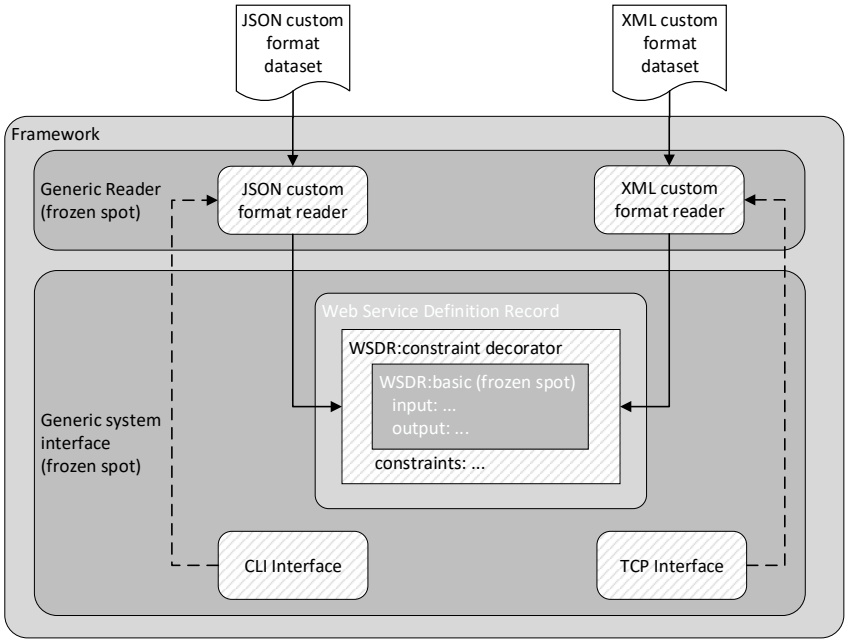

Figure 15: Framework instance: Readers

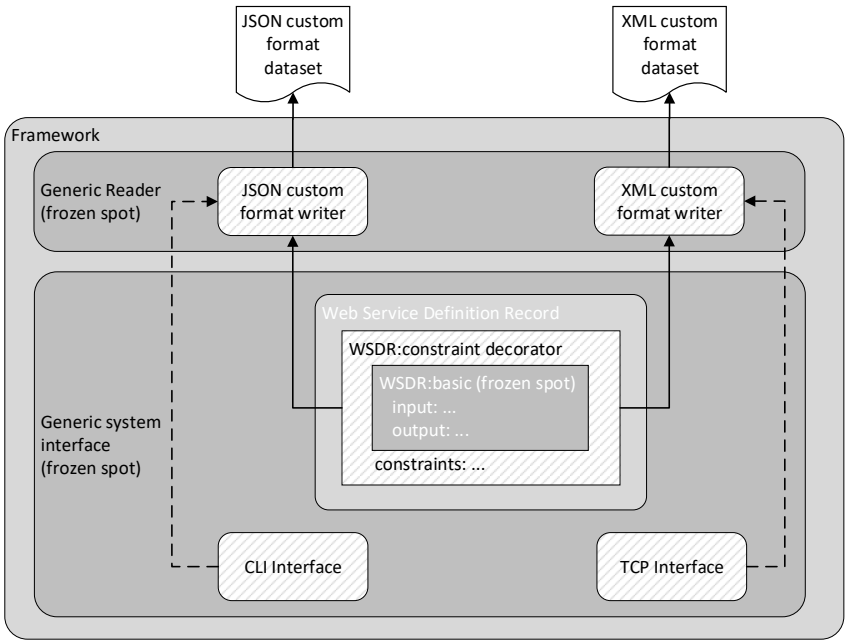

Figure 16: Framework instance: Writers

4.2.4 Service Records

The service records in our framework are illustrated in more detail with some examples of some hot spots in Figure 17. In Section 2.3.2, along with the basic Web service description record, we listed different definitions of a Web service that can extend the basic Web service definition record. In this case, we consider the basic Web service description record to be a frozen spot, and any other Web service description record, which must extend this definition by adding information to it, to be a hot spot. It is also possible to decorate a Web service description record that is not a basic Web service description record, allowing for arbitrarily complex Web service description records with many layers of additional decorating information.

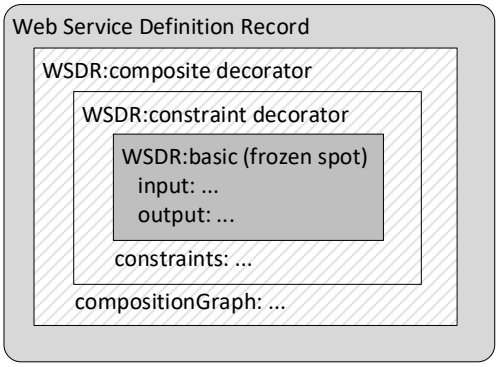

Figure 17: Framework instance: Service records

4.2.5 Translators

The translators in our framework are illustrated in more detail with some examples of some hot spots in Figure 18. The flow of information from the Web service definition records to the translator hot spots and vice versa is also illustrated, as well as how our system interfaces control the translators, with some examples. In this case, we consider the basic Web service description record to be a frozen spot, and any other Web service description record, which must extend this definition by adding information to it, to be a hot spot. Here are the two translator hot spots we have implemented:

To Constrained Web Service Description Record Translator: We created a translator hot spot that decorates the received Web service definition records with constraints. This translator is meant to create constrained web service definition records for testing algorithms that require constrained web service desription records as input. These constraints are randomly generated according to a given statistical distribution. The values, operators, and ontology types used are specified and then randomly chosen according to the previously mentioned statistical distribution.

Removing Decorating Information Service Description Record Translator: We created a translator hot spot that removes the additional information on a decorated Web service description record by extracting the Web service description record within. This translator simply goes through each Web service description record that was received and extracts the Web service definition record within, returning all the Web service description records that were extracted in this manner.

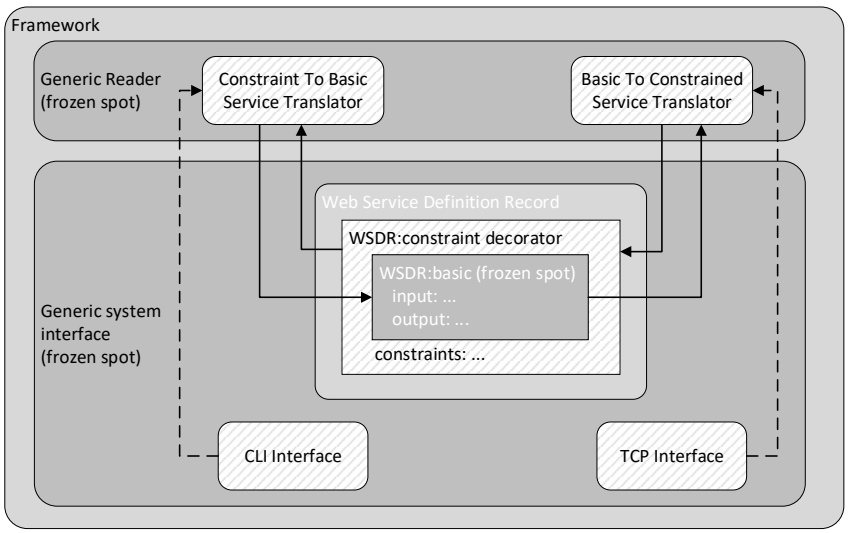

Figure 18: Framework instance: Translators

4.3 Instance Composition

To implement an instance of our framework, one must, at a minimum, obtain or implement one reader, writer, and service hot spot, along with at least one system

interface.

As a proof of concept and in order to test our design against the requirements laid out in Section 1.6, we have implemented a framework instance containing many framework hot spot instances as described in the previous sections.

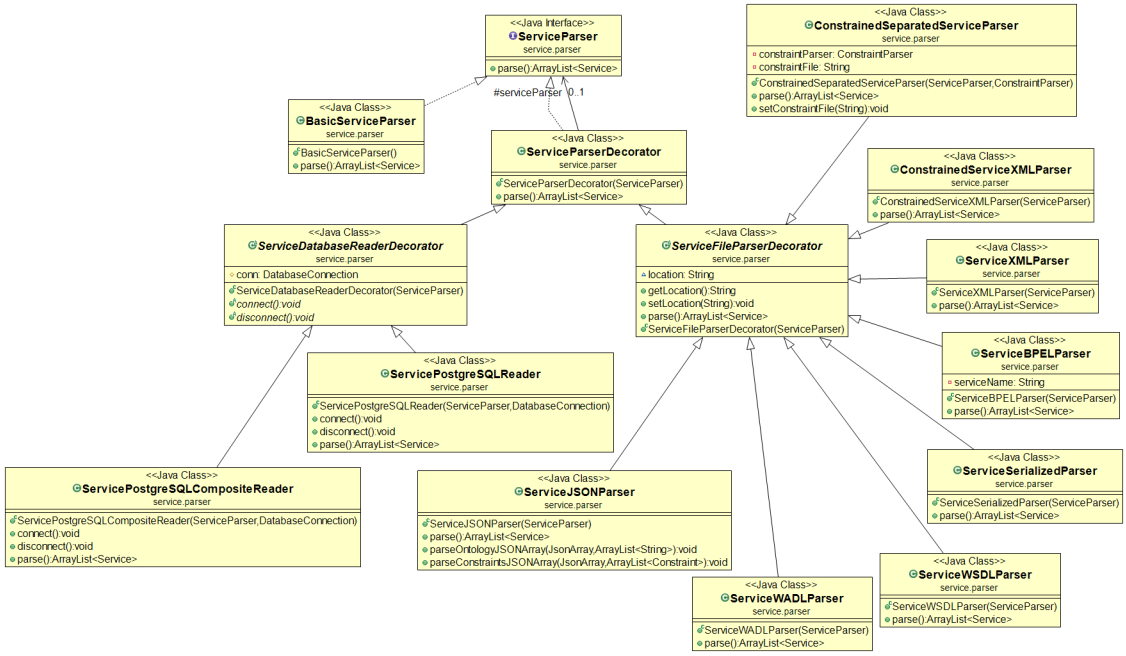

Figure 19: Reader decorator instances

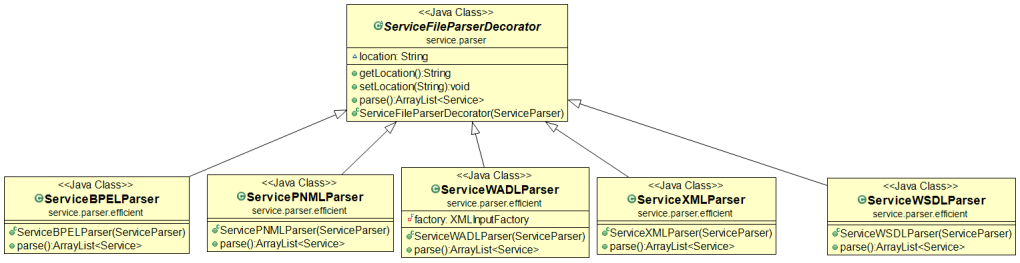

Figure 20: Reader decorator modules instances

We can see in Figure 19 and Figure 21 our class diagrams for our reader and writer decorators which we previously mentioned. In this case, we use inheritance to create two types of reader and writer decorators, namely those meant to read and write to and from databases and those meant to read and write to and from the file system. This distinction made sense, since both types of readers and writers are very different, but the decorators that use them all use some common fields or information. In the

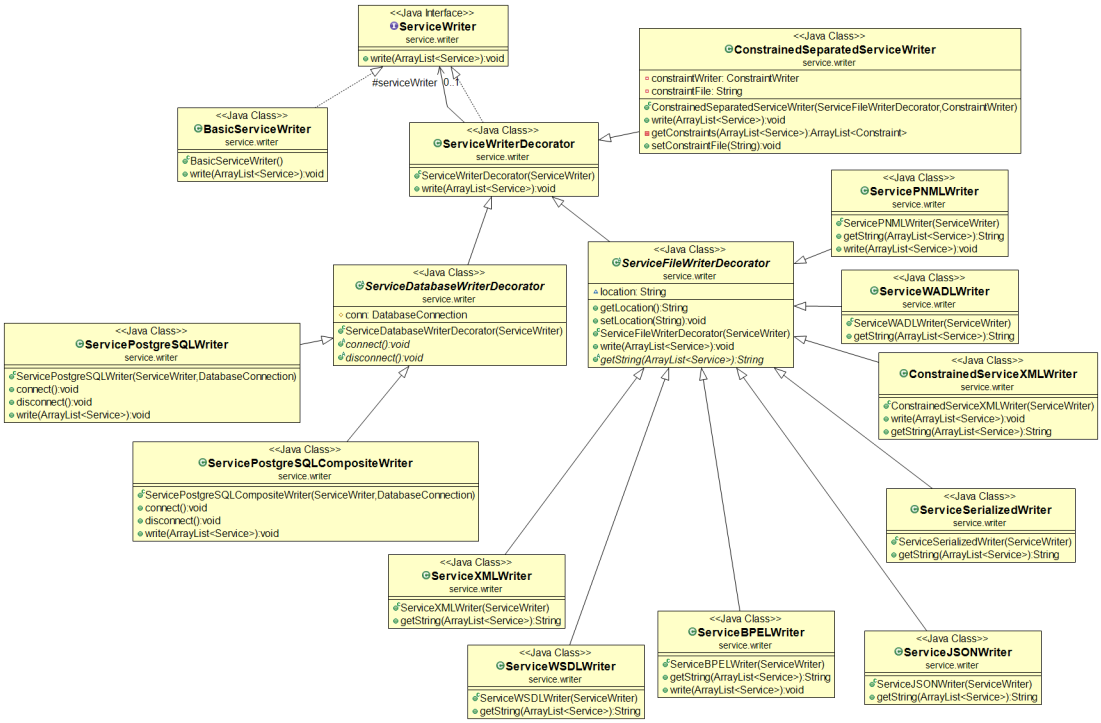

Figure 21: Writer decorator instances

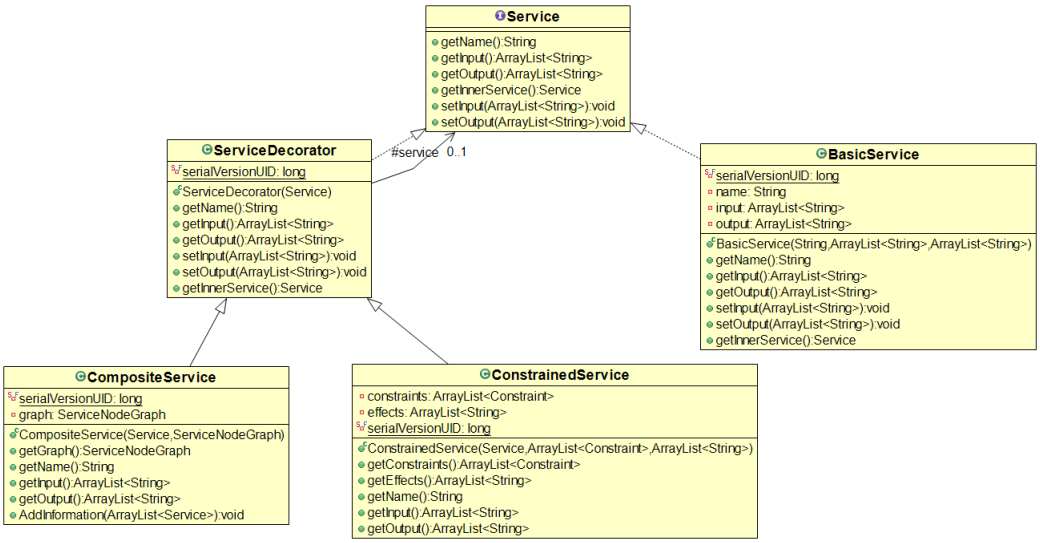

Figure 22: Service decorator instances

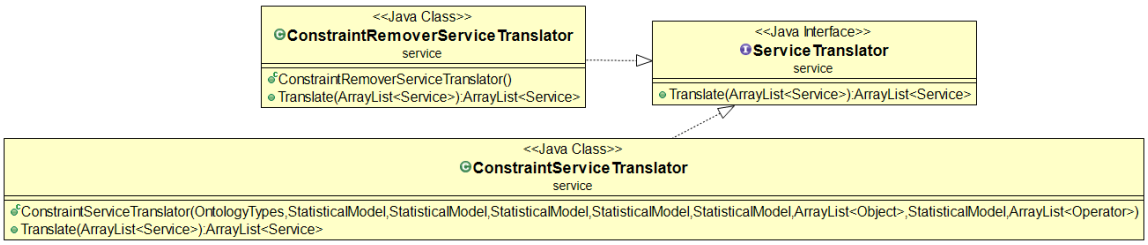

Figure 23: Translator instances

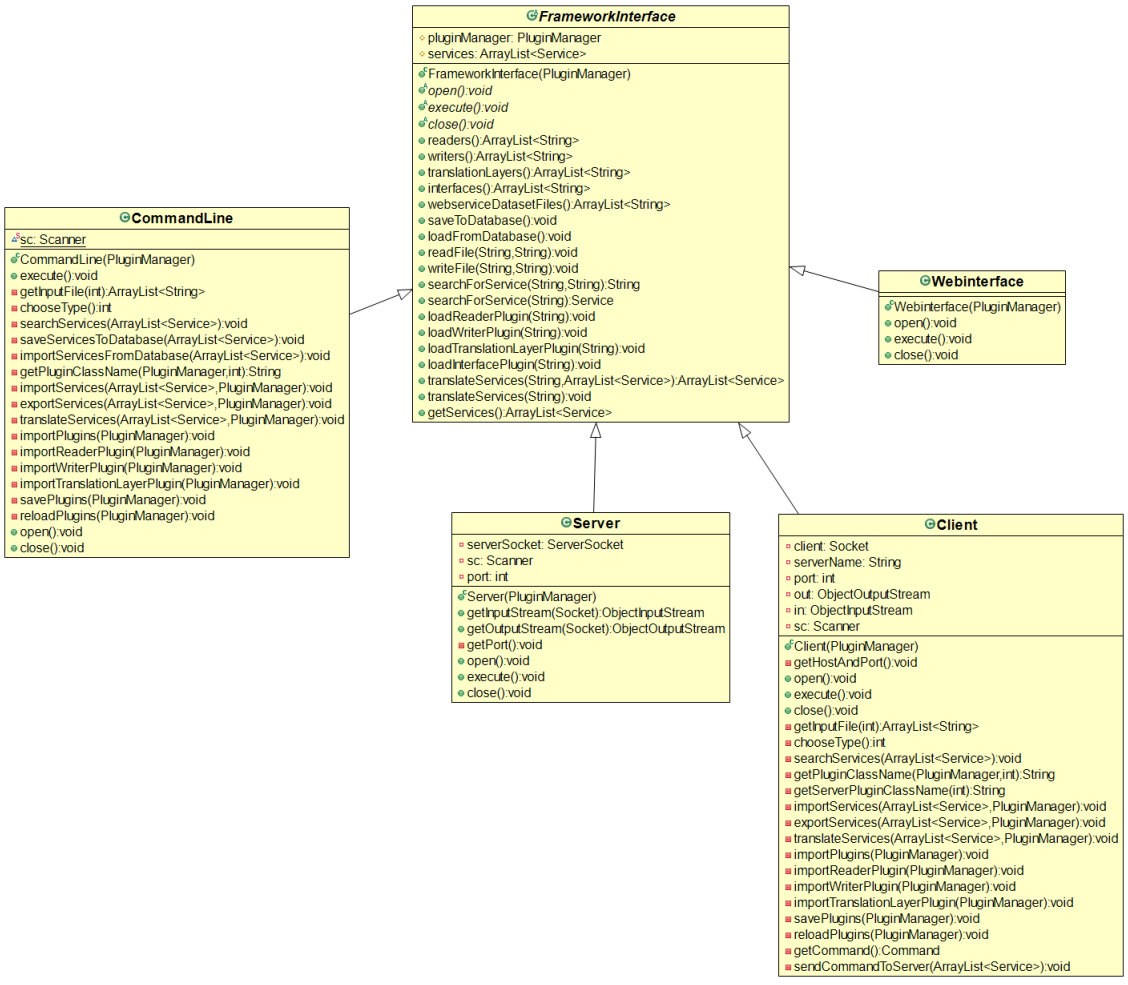

Figure 24: System interface instances

case of the readers that work with the file system, they all need a file location. In the case of the readers that communicate with a database, they all require a database connection object. In Figure 20 we have the class diagram describing our improved redesigned XML-based file readers that improve on the runtime of our initial XML-based file readers, which we discuss more in detail in Section 5.3.4.

In Figure 23, we illustrate the class diagram of the translators we have implemented. We have one translator which strips away constraint information by undecorating constrained Web service definition records. We also have a translator that decorates Web service definition records with constraint information that randomly adds constraints to Web service definition records according to a uniform distribution.

In Figure 24, we illustrate the class diagram of our system interfaces. We have our command line interface, our Web interface that controls an HTTP server that provides external access to our framework, and our server and client interfaces that communicate through a custom protocol built on top of TCP.

We can see in Figure 22 our class diagram for the decorators that we use to implement our Web service definition records. This decorator is much simpler, since all services share the same common information and only add to this.

4.4 Data Formats

Below are the details on the file formats and storage media we have chosen to read and write with this framework instance. For formats which we have defined ourselves, we have provided samples which include constraint information for the sake of completeness. For the purposes of writing and parsing, we have chosen to implement the following reader and writer hot spots:

Custom JSON Format: We have created a custom JSON format for testing purposes. It allows us to showcase the adaptability and extensibility we aim to achieve with our framework, since it allows us to demonstrate that a newly developed format can be accounted for in our system feasibly. It also allows us to more thoroughly

```
[{"name" : "mastercard",
  "input" : ["amount", "accountnumber"],
  "output" : ["balance",
   "travelrewardspoints"],
  "constraints" : [{"servicename" :
   "mastercard", "literalValue":
   "2000", "OntologyType" :
   "amount",
   "Operator": "<"}],
  "effects": ["balance"]},
  {"name" : "weathernetworkcanada"
    "input" : ["city", "country",
     "forecastlength"],
    "output" : ["averagetemperature",
     "mintemperature", "maxtemperature"],
    "constraints" : [{"servicename" :
     "weathernetworkcanada",
     "literalValue": "canada",
     "OntologyType" : "country",
     "Operator": "="}]}]
```

Figure 25: Custom JSON format output sample

verify that we can achieve the translation scenario we have described in Section 1.5.1

In Figure 25 is an example of `JSON` content with information on two separate Web services. We use a fictional "mastercard"service that allows users to send an amount to an account and receive a number of travel rewards points, as well as a fictional weather service that provides the average, minimum, and maximum temperature over the next specified number of days within a specified city. The first service has one constraint that the amount must be less than 2,000 dollars. The second service has one constraint that the country must be Canada, implying that this service only provides weather for Canadian cities.

Our custom `JSON` format consists of an array of service objects. Each of these objects has the following fields:

- `name`: A string representing the name of the service.
- `input`: An array of strings representing the `OntologyTypes` comprising the service's inputs.
- `output`: An array of strings representing the `OntologyTypes` comprising the

service's outputs.

- `constraints:` An array of objects representing the constraints which restrict this service. This object contains the following fields:
  - `servicename:` A string representing the name of the service this constraint applies to.
  - `literalvalue:` A string representing the value that restricts this constraint.
  - `OntologyType:` A string representing the ontology type that the value belongs to.
  - `operator:` A string representing the operator that governs the relationship between the `value` and the service and describes this restriction. This can be any of the following values: `=`, `<`, `>`, `=<`, or `=>`.
  - `effects:` An array of strings representing the `OntologyTypes` of the service's affected inputs.

Generator XML Format: We have created readers and writers that handle the custom XML format used by the WSC-gen Web service description generator [14].

Custom XML Format: We have adapted the XML format used by the WSC-gen Web service generator [14] to accommodate constraints and effects. It again allows us to showcase the adaptability and extensibility we aim to achieve with our framework, since it allows us to demonstrate again that a newly developed format can be accounted for in our system feasibly. It also allows us to still more thoroughly verify that we can achieve the translation scenario we have described in Section 1.5.1. Within Figure 26 is an example of XML content with information on services. We again use the same fictional "mastercard"service that allows users to send an amount to an account and receive a number of travel rewards points. The service has once again one constraint that the amount must be less than 2,000 dollars, and a new constraint that the account number must be greater than 0, since otherwise it cannot be an existing account number.

```
<?xml version="1.0" encoding="UTF-8"
standalone="no"?>
<services>
    <service name="mastercard">
        <inputs>
            <instance name="amount"/>
            <instance
             name="accountnumber"/>
       </inputs>
        <outputs>
            <instance name="balance"/>
            <instance
             name="travelpoints"/>
        </outputs>
        <constraints>
            <instance>
                <servicename
                name="mastercard"/>
                <literalvalue
                 name="2000"/>
                <type name="amount"/>
                <operator name="&lt;"/>
            </instance>
            <instance>
                <servicename
                 name="mastercard"/>
                <literalvalue name="0"/>
                <type
                 name="accountnumber"/>
                <operator name="&gt;"/>
            </instance>
        </constraints>
        <effects>
            <instance name="balance"/>
        </effects>
    </service>
</services>
```

Figure 26: Custom XML format output

Our custom XML format is defined with the root `services` node, which has no attributes. The children of this root element are a series of `service` elements. A `service` element has one attribute: `name`. The child nodes of a `service` node are as follows:

- `inputs`:This node represents the inputs for the parent `service` node. It has no attributes, and its children are a series of `instance` nodes with the `name` attribute, which represents the `OntologyType` that this input belongs to.
- `outputs`:This node represents the outputs for the parent `service` node. It has no attributes, and its children are a series of `instance` nodes with the `name` attribute, which represents the `OntologyType` that this output belongs to.
- `constraints`: This node represents the constraints which apply to this service. It has no attributes, and its children are a series of `instance` nodes with themselves no attributes. The children of these `instance` nodes are as follows:
  - `servicename`: A node representing the name of the service this constraint applies to. It has one attribute, `name`, whose value is the name of the service that this constraint applies to.
  - `literalvalue`: A node representing the value that restricts this constraint. It has one attribute, `name`, which contains a string representation of the value in question.
  - `type`: A node representing the ontology type that the value belongs to. It has one attribute, `name`, which contains a string representation of the type in question.
  - `operator`: A node representing the operator that governs the relationship between the `value` and the service that describes this restriction. This can be any of the following values: `=`, `<`, `>`, `=<`, or `=>`. It has one attribute, `name`, which contains a string representation of the operator in question, using `HTML` codes when necessary.

– `effects`: This node represents the effects for the parent `service` node. It has no attributes, and its children are a series of `instance` nodes with the `name` attribute, which represents the `OntologyType` that this effect applies to.

Serialized Java objects: Serialized JAVA objects are easy to use with JAVA. In the cases for which a specific algorithm is using service descriptions stored as Java objects in memory, this option would be the easiest and most convenient to use.

WSDL: We included this format because it is a popular standard found throughout the Web. We have built a parser which is capable of parsing the WSDL output produced by the WSC-Gen Web service generator [14].

BPEL: We included this format because it is one of the standards used to describe composite Web services. It is a format produced by the WSC-Gen Web service generator [14]. It also complements any research on composite Web services and showcases the flexibility of our system due to the complex nature of composite Web services.

WSLA: The WSC-Gen Web service generator [14] produces files of this type, and it is a popular format. We chose this file type to showcase our ability to read and write constraint information to and from disk.

WADL: Also a popular format used on the Web, though it is not included as part of the output produced by the WSC-Gen Web service generator [14]. For this reason, we included this file type to showcase the flexibility of our repository to accept new and varying formats.

Separated Parser and Writer: We have created reader and writer hot spots that read and write constrained Web service records in the case where basic Web service records are saved in one location and the applicable constraints are saved in a separate file. These read or write both the basic Web service definition records and the constraints to the different locations simultaneously. The reader then decorates the basic Web service Web definition records with the constraints that apply to each

service. This is useful in the case where Web service definition records are defined in WSDL and WSLA files together and must both be read or written to reconstruct or maintain the constrained Web service definition records.

PostgreSQL: As a popular database, this demonstrates our ability to read Web service definition records not only from the file system, but also from other media as well. We have hot spots that can read or write both constrained and composite Web service definition records.

4.5 Summary

This chapter described the framework instance we created to fulfill our motivation scenarios, as mentioned in Section 1.5. We explained the relationship between all of our hot spots in Section 4.2. We described our interfaces in Section 4.2.1, our readers in Section 4.2.2, our writers in Section 4.2.3, our Web service definition records in Section 4.2.4, and our translators in Section 4.2.5. The specific details on the hot spots we have created were discussed in Section 4.3. The next chapter describes the tests we designed and executed using the hot spots described in this chapter in order to demonstrate that we have met the objectives and requirements that we had stated in Chapter 1.

Chapter 5

Evaluation

In this chapter, we evaluate our solution by checking whether or not our goals previously mentioned in Section 1.7 were attained and whether the motivation scenarios were shown to be achievable by use of our framework implementation, as instantiated in the previous chapter. Further in this section we lay out the tests used to perform this evaluation along with the results of these tests. For all tests in this chapter, we first had to create an instance of our framework as described in Chapter 4.

5.1 Overview

In Section 5.2, we describe the various methods that may be used to extend our solution. In Section 5.2.1, we describe how one can manually instantiate a hot spot one has created oneself. In Section 5.2.2, we describe how one can use our plugin manager to import a hot spot that one has created oneself. In Section 5.2.3, we describe how one can manually instantiate a hot spot that was included in a library. In Section 5.2.4, we explain how one can import a hot spot included in a library using the plugin manager. In Section 5.2.5, we explain how one can create a program external to our framework that interfaces with our framework. In Section 5.2.6, we explain how we can allow a pre-existing solution that has no knowledge of our framework to communicate with our framework. In Section 5.2.7, we describe how our framework can be used as a library to build a solution.

In Section 5.3, we describe the tests we designed to verify that we can achieve the motivation scenarios that we had established in Chapter 1. We describe how we can translate a dataset in Section 5.3.1, how we can achieve time scalability in Section 5.3.2, how we can combine different datasets in Section 5.3.3, how we can achieve space scalability in Section 5.3.4. We then cover how we can extend our framework, starting with creating a hot spot in Section 5.3.5, importing a plugin in Section 5.3.6, how a researcher can use a public instance of our solution in Section 5.3.7, and how a local instance can connect to a public instance in Section 5.3.8. We then cover how we can use translated data in an algorithm in Section 5.3.9.

In Section 5.4, we describe how we designed tests that verify that we have met the quality requirements we have previously mentioned. We verify that we have met our goal of interoperability in Section 5.4.1, adaptability in Section 5.4.2, extensibility in Section 5.4.3, scalbility in Section 5.4.4, and usability in Section 5.4.5.

5.2 Framework Integration Methods

We described the hot spots we created for our framework in Section 4.3, but we have not described how a hot spot is introduced into our framework, or, more broadly, all the possible ways one may use our framework to achieve their goals. We have identified seven different ways to achieve this, each being useful in different situations. We describe them in the following sections.

5.2.1 Manually Referencing a Custom Class

One may wish to create their own class in our Java project to create a new hot spot which extends one of the frozen spots in our framework. Each of our frozen spots exists in its own namespace with classes that implement this frozen spot. After such a hot spot is created, it may be manually referenced where appropriate. For example, reader and writer hot spots may be manually instantiated in the `execute` function of a system interface hot spot, or a Web service definition record hot spot may be

manually instantiated within a reader hot spot. This is useful if one requires precise, immutable, fine-grained control over the hot spots one has created. The following are the hot spots we have embedded in the framework instances in our tests in this manner:

- Our WSDL reader included in the framework instances described in Section 5.3.3, Section 5.3.4, and Section 5.3.7
- Our WADL reader included in the framework instances described in Section 5.3.1, Section 5.3.2, Section 5.3.3, and Section 5.3.4
- Our custom WSC-gen XML format reader included in the framework instances described in Section 5.3.1 and Section 5.3.2
- Our custom JSON format reader included in the framework instances described in Section 5.3.1 and Section 5.3.9
- Our PostgreSQL reader included in the framework instances described in Section 5.3.4
- Our WSDL writer included in the framework instances described in Section 5.3.1, Section 5.3.2, Section 5.3.3, and Section 5.3.7
- Our WADL writer included in the framework instances described in Section 5.3.7 and Section 5.3.9
- Our custom XML format writer included in the framework instances described in Section 5.3.1 and Section 5.3.7
- Our custom JSON format writer included in the framework instances described in Section 5.3.7
- Our PostgreSQL writer included in the framework instances described in Section 5.3.3 and Section 5.3.4
- Our combined WSDR and constraint writer included in the framework instances described in Section 5.3.1, Section 5.3.2, and Section 5.3.3

- Our constrained Web service definition record hot spot included in the framework instances described in Section 5.3.1, Section 5.3.2, Section 5.3.3, and Section 5.3.4
- Our translator that randomly adds constraints to Web service definition records included in the framework instances described in Section 5.3.1, Section 5.3.2, and Section 5.3.9
- Our translator that strips away the outermost layer of decorating information included in the framework instances described in Section 5.3.1.

5.2.2 Importing a Custom Class as a Plugin

Similarly to what is described in Section 5.2.1, one may have created a custom hotspot in our Java code, but not wish to manually instantiate this hot spot. In such a case, one may use the plugin manager described in Section 3.6 to import the class as a plugin and delegate the responsibility of creating an instance of this class to the plugin manager. To do this, one must tell the plugin manager to import the class as a plugin using the class's full name, including its namespace. For example, to import the hot spot for our parser for our custom JSON format as a plugin, one would use the following name: `service.parser.ServiceJSONParser`. The following are the hot spots we have embedded in the framework instances in our tests in this manner:

- Our CLI system interface, included in the framework instances described in Section 5.3.1, Section 5.3.2, Section 5.3.3, Section 5.3.4, and Section 5.3.9
- Our TCP client system interface included in the framework instances described in Section 5.3.8
- Our TCP server system interface included in the framework instances described in Section 5.3.8
- Our HTTP server system interface included in the framework instances described in Section 5.3.7

5.2.3 Manually Referencing a Class in a Library

We mention in Section 3.6 that a researcher may want to download a JAR file with the classes that implement some hot spots which they wish to use. In this case, however, they may wish to manually create an instance of the hot spots within and bypass the plugin manager altogether. Similar to what was mentioned in Section 5.2.1, this researcher may wish to do this to maintain fine-grained control over the hot spots contained in a JAR file. This is possible as long as the researcher has properly added the JAR file to the Java code and properly added it to the list of referenced libraries. We have imported the Java serialized object writer in our tests in this manner in the framework instance described in Section 5.3.7.

5.2.4 Importing a Class in a Library as a Plugin

We mention in Section 3.6 that a researcher may want to download a JAR file with the classes that implement some hot spots which they wish to use. In this case the researcher wishes to use the plugin manager to import the hot spots contained within the JAR file, similarly to what was described in Section 5.2.2. The researcher will again wish to delegate the responsibility of creating an instance of the Java class to the plugin manager. As before, one must tell the plugin manager to import the class as a plugin using the class's full name, including its namespace. For example, to import the hot spot for our HTTP system interface as a plugin from a JAR file, one would use the following name: `userinterface.web.Webinterface`. We have imported the WSC-gen XML format writer in our tests in this manner in the framework instance described in Section 5.3.1 and Section 5.3.2.

5.2.5 Writing an External Program that Interfaces with a Framework Instance

The system interface hot spots described in Section 4.2.1 allow our framework to communicate with a user or external program. In this particular case, we consider the ways that an external program can communicate with our framework. For

example, our HTTP interface allows an external program to communicate with a public instance of our framework available on a public server through HTTP requests and responses. In the case of our TCP client and server interfaces, the external program is another instance of our framework, namely our TCP client interface, mentioned in Section 5.3.8, where the communication is done through a custom protocol built over TCP described in Section 4.2.1. Additionally, we have created a Python script that communicates with our HTTP interface, used to interface with a framework instance, as described in Section 5.3.7.

5.2.6 Interfacing a Pre-existing Program with a Framework Instance

In other cases, the communication must be indirect and handled manually. For example, in Section 5.3.9, we describe how we allow the SoapUI [45] program to read incompatible data by first translating it with our framework. This program can only read the framework's output that is correctly formatted according to the needs of SoapUI. Therefore, when the translation is finished, the files produced must be manually fed to SoapUI for SoapUI to parse the data. It can therefore be acknowledged that SoapUI reads data that is incompatible to it, but only in an indirect way which requires manual intervention. We have connected the SoapUI tool [45] to our framework in this manner used to extend the framework instance described in Section 5.3.9.

5.2.7 Using a Framework Instance as a Library

A JAR of our framework can be used to access the modules within in one's own Java program by simply including said JAR in a Java project and using the modules defined within. This offers one the most freedom to use our modules without any restrictions. However, this comes at a price, as this is the most difficult method of using our framework that requires the most technical knowledge, including the details of the modules in our framework. This method also means that, in this particular

case, our solution is no longer a framework as defined in Section 2.2.3, due to the lack of a guaranteed inversion of control. Our solution then becomes much closer to a dynamic library as defined in Section 2.2.2. We mention this method of using our solution in the interest of completeness and to demonstrate that our solution is flexible enough to accommodate such a use case if desired. Another student in our research group has leveraged and extended the functionality of our framework as this way to produce algorithms related to service composition [57].

5.3 Realization of Motivation Scenarios

In Section 1.5, we listed a number of motivation scenarios that highlight some problems that are not solved by existing research solutions. For each of them, we devised a test to determine whether or not our framework allows a user to achieve this motivation scenario. For each of these tests, we assume that an implementation of our framework is available as described in Chapter 3. We start the presentation of all our demonstration scenarios from a bare framework and explain the series of steps that need to be done to create a framework instance that provides a solution to the scenario, thereby describing the usability of our solution for each required usage scenario.

5.3.1 Translating Datasets

We discussed in Section 1.5.1 how a researcher might want to translate a dataset to a different format, Web service description model, or both. We tested that our repository can perform this task. Figure 27 shows the steps we followed to build the framework instance used in this test.

This test was performed in two parts. Both demonstrate translating from one file format to another, but the first also demonstrates the ability to add additional decorating information, and the second instead demonstrates the ability to strip away additional decorating information.

1. Create a reader hot spot that reads WADL files. We manually created and added this hot spot to our code as described in Section 5.2.1

2. Create a reader hot spot that reads our custom JSON format. We manually created and added this hot spot to our code as described in Section 5.2.1

3. Create a writer hot spot that writes the custom XML format used by the WSC-gen Web service generator [14] as decribed in Section 4.3. We created this as a custom class, then placed it in a library to be imported as a plugin as described in Section 5.2.4

4. Create a writer hot spot that writes WSDL files. We manually created and added this hot spot to our code as described in Section 5.2.1

5. Create a writer hot spot that takes another writer hot spot as a parameter along with a constraint writer object. We use this to write services in WSDL files and write the applicable constraints in WSLA files. We manually created and added this hot spot to our code as described in Section 5.2.1

6. A hot spot that decorates the basic Web service definition record with constraints. We manually created and added this hot spot to our code as described in Section 5.2.1

7. Create a translator hot spot that translates constrained Web service definition records to unconstrained Web service definition records by stripping away additional decorating information to extract the basic Web service description record within. We manually created and added this hot spot to our code as described in Section 5.2.1

8. Create a translator hot spot that translates basic Web service definition records to constrained Web service definition records by randomly creating constraints for each Web service description record according to a uniform distribution. We manually created and added this hot spot to our code as described in Section 5.2.1

9. Create a command line system interface hot spot that allows users to control the reader, writer, and translator hot spots we have created. We created this as a custom class, then imported it as a plugin as described in Section 5.2.2

Figure 27: Steps to achieve "Translating Datasets"

First Part:

We built our own dataset by gathering 3 WADL files obtained arbitrarily from the internet. We then took this dataset, and read it in. We then decorated these Web service definition records with constraint information using our translator. Lastly, we outputted these decorated Web service definition records to the disk in a WSDL and WSLA file. We illustrate this test in Figure 28.

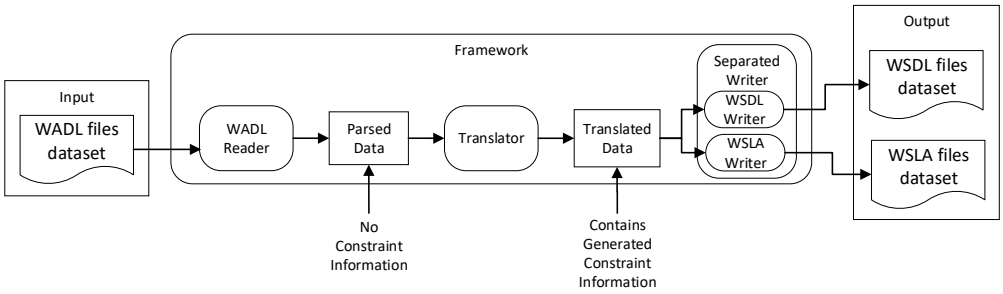

Figure 28: Translating datasets from WADL to WSDL and WSLA

Second Part:

To demonstrate our ability to also remove information that is unnecessary, we read in a dataset equivalent to the one mentioned earlier, written in a JSON file, stripped it of its constraint information using another translator , and wrote the result in an XML file using our custom XML format's writer. We illustrate this test in Figure 29.

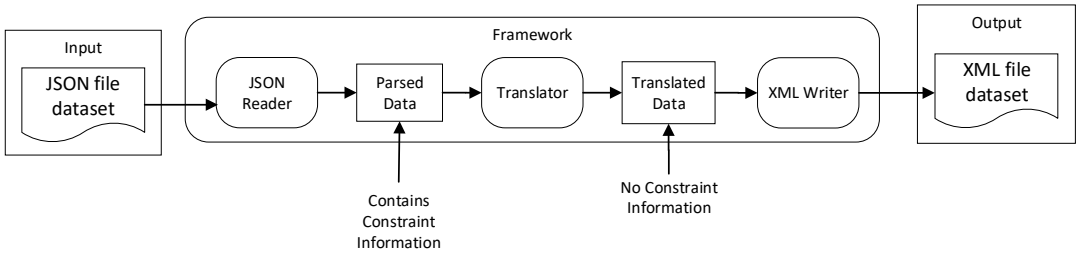

Figure 29: Translating datasets from JSON to XML

5.3.2 Time Scalability

We discussed in Section 1.5.3 how we rebuilt a series of modules to suit our needs after we found that the initial modules were insufficiently optimized for our needs. While building the initial test mentioned in Section 5.3.1 we found that the modules we were using for parsing the XML-based data formats involved were too inefficient to be usable. Upon further investigation, we found that the XML parsing library which we were using in these modules was inefficient and only meant for much smaller files. We found an efficient replacement library, rewrote the offending modules with it, and ran the test again. Figure 30 shows the steps we followed to build the framework instance used in this test.

We performed the same test described in the first part of Section 5.3.1. More precisely, we built our own dataset by gathering 3 WADL files obtained arbitrarily from the internet. We then took this dataset, and read it in. We then decorated these Web service description records with constraint information using our translator. Lastly, we outputted this data to the disk in a WSDL and WSLA file.

We quickly found that this test was running very inefficiently. Indeed, we originally translated the WADL dataset to WSDL and WSLA files in 118.219 seconds. After discovering that the module that was responsible for our inefficiency was the reader that read the WADL dataset, we created a replacement module that used a much more efficient library. We clearly show in Section 5.3.1 that we have achieved a significant improvement on this runtime with this new, more efficient reader hot spot, performing the translation in 0.661 seconds. Thus, we have shown that it is possible to include a module with greater time scalability than another module with comparable functionality. In theory, it is possible to exploit even more time scalability by adding a module that exploits a technology or algorithm that offers still more time scability.

More generally, we have shown that our framework's design does not inherently limit one from creating modules with the appropriate level of time scalability for a particular task, such as the one described in the test defined in Section 5.3.1.

1. Create a reader hot spot that reads WADL files. We manually created and added this hot spot to our code as described in Section 5.2.1

2. Create a writer hot spot that writes WSDL files. We manually created and added this hot spot to our code as described in Section 5.2.1

3. Create a writer hot spot that takes another writer hot spot as a parameter along with a constraint writer object. We use this to write services in WSDL files and write the applicable constraints in WSLA files. We manually created and added this hot spot to our code as described in Section 5.2.1

4. A hot spot that decorates the basic Web service definition record with constraints. We manually created and added this hot spot to our code as described in Section 5.2.1

5. Create a translator hot spot that translates basic Web service definition records to constrained Web service definition records by randomly creating constraints for each Web service description record according to a uniform distribution. We manually created and added this hot spot to our code as described in Section 5.2.1

6. Create a command line system interface hot spot that allows users to control the reader, writer, and translator hot spots we have created. We created this as a custom class, then imported it as a plugin as described in Section 5.2.2

7. After determining that our WADL reader was inefficient, we replaced it with a mode efficient reader hot spot that took advantage of a more efficient XML parsing library. We manually created and added this hot spot to our code as described in Section 5.2.1, as we did with the previous version of this reader.

Figure 30: Steps to achieve "Time Scalability"

5.3.3 Combining Datasets

We discussed in Section 1.5.2 how a researcher might want to combine different datasets. We tested that our repository can perform this task. Figure 31 shows the steps we followed to build the framework instance used in this test.

For this test, we built our own dataset by gathering 3 WADL files obtained arbitrarily from the internet. We then took this dataset, read it in, read in a portion of Laleh's WSDL dataset [26]. We then decorated these Web service definition records with constraint information using our translator. Lastly, we outputted the combined results to the disk in a series of WSDL and WSLA files. We then stored this information to a database as well as a filesystem to demonstrate that our system can be scalable by leveraging a module which provides scalability as a feature, while also being flexible and allowing the user to choose which module they would prefer to use. Note that we used the previously mentioned modules with improved efficiency for this test when applicable. We illustrate this test in Figure 32

5.3.4 Space Scalability

In the implementation used in this test, any hot spot that handles data stored on the filesystem will do, such as WSLA. Afterwards, a hot spot that communicates with a database that can handle a large amount of data, such as PostgreSQL, is required. We also implemented a basic command line system interface hot spot to allow us to run these tests.

We discussed in Section 1.5.4 how one researcher might find a specific module to be too restrictive in terms of the size of the data that can be read or written. We mentioned that if a module was unsatisfactory, it could be swapped out for a better one that could meet the needs of the user. We tested that our repository can perform this task. Figure 33 shows the steps we followed to build the framework instance used in this test.

In our case, to show that a user could replace a module with a new one that would allow them to store and manage more data, we created reader and writer

1. Create a reader hot spot that reads WADL files. We manually created and added this hot spot to our code as described in Section 5.2.1
2. Create a reader hot spot that reads WSDL files. We manually created and added this hot spot to our code as described in Section 5.2.1
3. Create a writer hot spot that writes WSDL files. We manually created and added this hot spot to our code as described in Section 5.2.1
4. Create a writer hot spot that takes another writer hot spot as a parameter along with a constraint writer object. We use this to write services in WSDL files and write the applicable constraints in WSLA files. We manually created and added this hot spot to our code as described in Section 5.2.1
5. Create a writer hot spot that writes constrained Web service definition records to our PostgreSQL database. We manually created and added this hot spot to our code as described in Section 5.2.1
6. A hot spot that decorates the basic Web service definition record with constraints. We manually created and added this hot spot to our code as described in Section 5.2.1
7. Create a translator hot spot that translates basic Web service definition records to constrained Web service definition records by randomly creating constraints for each Web service description record according to a uniform distribution. We manually created and added this hot spot to our code as described in Section 5.2.1
8. Create a command line system interface hot spot that allows users to control the reader, writer, and translator hot spots we have created. We created this as a custom class, then imported it as a plugin as described in Section 5.2.2

Figure 31: Steps to achieve "Combining Datasets"

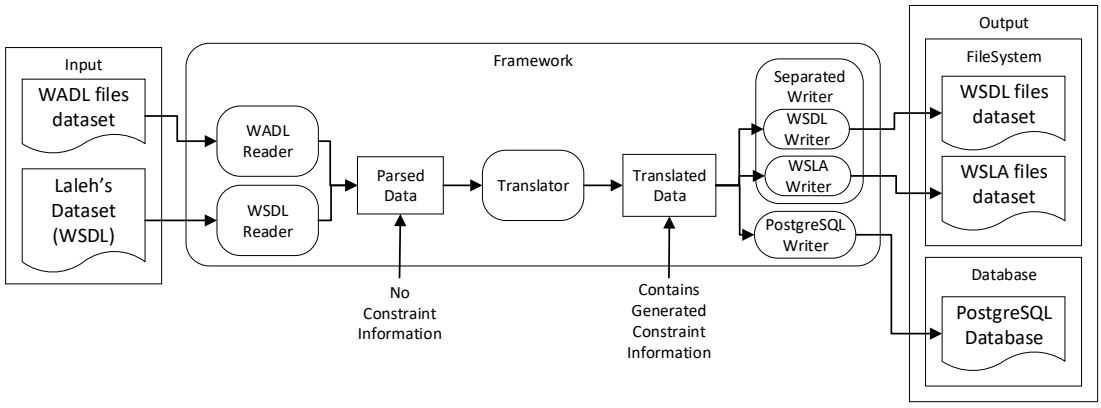

Figure 32: Combining WADL and WSDL datasets

1. Create a reader hot spot that reads WADL files. We manually created and added this hot spot to our code as described in Section 5.2.1
2. Create a reader hot spot that reads WSDL files. We manually created and added this hot spot to our code as described in Section 5.2.1
3. Create a reader hot spot that reads constrained Web service definition records from our PostgreSQL database. We manually created and added this hot spot to our code as described in Section 5.2.1
4. Create a writer hot spot that writes constrained Web service definition records to our PostgreSQL database. We manually created and added this hot spot to our code as described in Section 5.2.1
5. A hot spot that decorates the basic Web service definition record with constraints. We manually created and added this hot spot to our code as described in Section 5.2.1
6. Create a translator hot spot that translates basic Web service definition records to constrained Web service definition records by randomly creating constraints for each Web service description record according to a uniform distribution. We manually created and added this hot spot to our code as described in Section 5.2.1
7. Create a command line system interface hot spot that allows users to control the reader, writer, and translator hot spots we have created. We created this as a custom class, then imported it as a plugin as described in Section 5.2.2

Figure 33: Steps to achieve "Space Scalability"

modules that could connect to a PostgreSQL database. A database can manage data much more efficiently than a filesystem, and thus allow for more data to be managed. These module allows a user to add any number of services to the database, as well as retrieve all services within the database and search for specific services by name. In Section 5.3.3, we mentioned that we used our PostgreSQL database to demonstrate space scalability. Indeed, we demonstrate that it is possible to include a module that offers greater space scalability by leveraging different technologies. In theory, it is possible to exploit even more space scalability by adding a module that exploits a technology that offers still more space scability.

In the case of this specific test, we designed one very similar to the one described in Section 5.3.3. We created a set of constrained Web service definition records by reading in the same WADL and WSDL files and decorating the Web service definition records within with constraint information using the same translator. We then saved all these Web service definition records to our database. We illustrate this in Figure 34. Afterwards, to demonstrate that our database reader hot spot, we flushed all the current Web service definition records from the command line system interface's list of Web service definition records, and then read all the constrained Web service definition records from our database that we had previously saved there, placing them in the list of Web service definition records managed by our command line system interface. We illustrate this in Figure 35

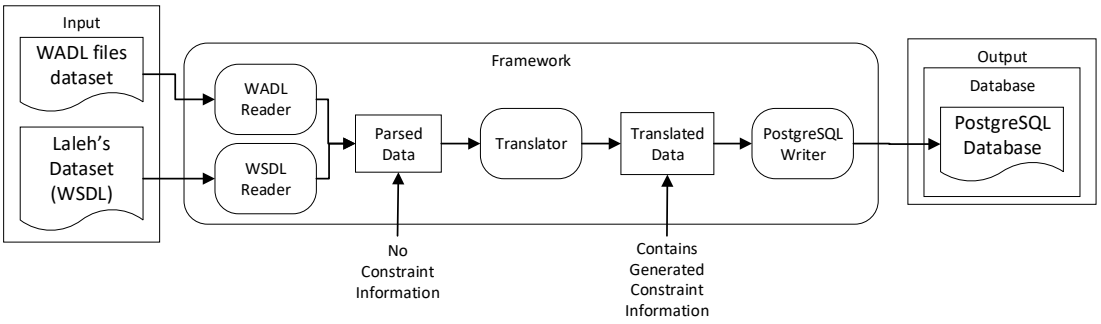

Figure 34: Writing services to our database

More generally, we have shown that our framework's design does not inherently

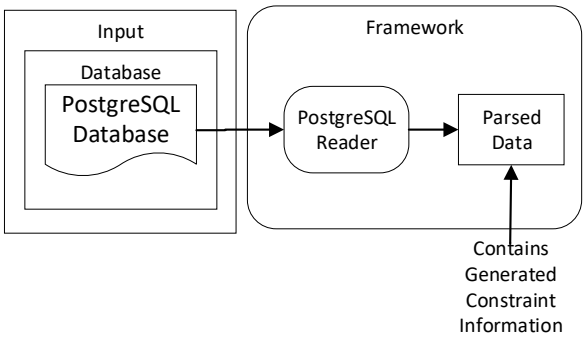

Figure 35: Reading services from our database

limit one from creating modules with the appropriate level of space scalability for a particular task.

5.3.5 Creating a New Hot Spot

In the implementation used in this test, no prior hot spots are necessary, since we are demonstrating the ability to add a new hot spot to our framework by manually adding it to existing code.

We mentioned that we needed a way to optimally extend our framework in Section 1.5.6. One can extend the functionality of our framework by creating a new hotspot directly integrated into our code by modifying our code base. This is explained in Section 5.2.1. We have included many hot spots in our framework this way. For example, this his how we included the efficient parsers that parse XML-based formats, such as the more efficient of our two WADL parsers mentioned in Section 5.3.2.

5.3.6 Importing a Plugin

In the implementation used in this test, no prior hot spots are necessary, since we are demonstrating the ability to add a new hot spot to our framework through the plugin manager.

We created a system to handle custom plugins in the executable we created to

access our framework's functionality. They allow a user to use a newly created hotspot in the executable without needing to modify our code, re-compiling our executable, or even writing any code at all. The modules can even be written by someone else and simply imported by the user. We mentioned this functionality in Section 1.5.7. We used this functionality to import some of the hot spots we have created. For example, all of our system interfaces are imported as plugins. This functionality is explained in Section 5.2.4 and illustrated in Figure 10.

One can also combine custom modules and imported plugins, both mentioned previously.

5.3.7 Interfacing with a Server Framework Instance

We mention that we need a way to interface with a publicly available instance of our framework in Section 1.5.8. In this case, one does not wish to work with our code base at all, and we offer the option of accessing a publicly accessible instance of our framework through HTTP. Figure 36 shows the steps we followed to build the framework instance used in this test.

One can use our HTTP system interface to access our data, download it to their local machine, and process it there with their own program. This eliminates the need to have any knowledge of our framework or code and provides a simplified interface to access easily. This is is explained in Section 5.2.5 and illustrated in Figure 12.

We tested that one can use our HTTP system interface by writing a Python script that accesses said system interface. First, our Python script sends out an HTTP Post request that sends a WSDL file defining some services to our HTTP system interface. Our HTTP system interface parses this WSDL file, extracts the services, and saves them to our database. Then, our Python script sends several HTTP requests to our HTTP system interface, each one whose response contains all the Web service description records currently stored in the database, namely those we have uploaded, in one of our previously described supported formats. This is done by using the reader hot spots we have previously mentioned to generate the appropriate strings to send in the HTTP responses. In this way, we cover the following formats: our custom

1. Create a reader hot spot that reads WSDL files. We manually created and added this hot spot to our code as described in Section 5.2.1

2. Create a writer hot spot that writes constrained Web service definition records in our custom XML file format. We manually created and added this hot spot to our code as described in Section 5.2.1

3. Create a writer hot spot that writes constrained Web service definition records in our custom JSON file format. We manually created and added this hot spot to our code as described in Section 5.2.1

4. Create a writer hot spot that writes Web service definition records in WADL. We manually created and added this hot spot to our code as described in Section 5.2.1

5. Create a writer hot spot that writes Web service definition records in WSDL. We manually created and added this hot spot to our code as described in Section 5.2.1

6. Create a writer hot spot that writes Web service definition records as serialized Java objects. We manually created and added this hot spot to our code as described in Section 5.2.1

7. Create an HTTP system interface hot spot that allows users to control the reader, writer, and translator hot spots we have created. We created this as a custom class, then imported it as a plugin as described in Section 5.2.2

8. Create as an external program that interfaces with our framework as described in Section 5.2.5. Specifically, we created a python script described further in this section.

Figure 36: Steps to achieve "Interfacing with Our Public Server"

JSON and XML formats, WADL, WSDL, and our serialized Java format. Note that in the case where a format requires constraints, since all the uploaded Web service definition records have none, the constraint information will be left empty to signify that there are no constraints.

5.3.8 Connecting a Local Instance with a Server Instance

In the implementation used in this test, at least one reader and writer are necessary, along with at least one translator. We also implemented two interface hot spots, namely one server and one client that communicate over TCP, to allow us to run these tests. This is a particularly interesting case as we have created as an external program that interfaces with our framework as described in Section 5.2.5, where this external program is another instance of our framework.

We mention in Section 1.5.9 that we needed a way to combine all the previously mentioned methods of interfacing with our framework (Section 5.3.5, Section 5.3.6, and Section 5.3.7). By connecting a private instance with our public instance, we can effectively create a cluster that contains the data and modules of several framework instances. We created a pair of client and server interfaces that communicate through TCP to achieve this, described in Section 4.2.1. This is described in Section 5.2.5 and illustrated in Figure 14.

5.3.9 Running an Algorithm on Translated Data

We mentioned in Section 1.5.5 that a researcher might want to take an existing implementation of an algorithm and wish to feed it a dataset which is in a format incompatible with this implementation's expected input format. We tested that this is achievable using our framework. Figure 37 shows the steps we followed to build the framework instance used in this test.

For this test, we built our own dataset by writing a file in our custom JSON format by hand, which contains two constrained Web service description records. We used our reader hot spot that reads our custom JSON format, read in these two services

1. Create a reader hot spot that reads files in our custom JSON format. We manually created and added this hot spot to our code as described in Section 5.2.1

2. Create a writer hot spot that writes WADL files. We manually created and added this hot spot to our code as described in Section 5.2.1

3. Create a hot spot that decorates the basic Web service definition record with constraints. We manually created and added this hot spot to our code as described in Section 5.2.1

4. Create a translator hot spot that translates constrained Web service definition records to unconstrained Web service definition records by stripping away additional decorating information to extract the basic Web service description record within. We manually created and added this hot spot to our code as described in Section 5.2.1

5. Create a command line system interface hot spot that allows users to control the reader, writer, and translator hot spots we have created. We created this as a custom class, then imported it as a plugin as described in Section 5.2.2

6. We obtained a program, which we interfaced with our solution as described in Section 5.2.6. Specifically, we chose the SoapUI tool [45]

Figure 37: Steps to achieve "Running an Algorithm on Translated Data"

from the file, and sent them through our translator hot spot to remove constraint information. We then took these two basic Web service description records and then wrote them to a WADL file using our WADL writer hot spot. This file was then sent to the SoapUI tool [45] to generate mock REST services. We illustrate this test in Figure 38. We imagine that a researcher could want to create mock REST services representing the services described in a custom format incompatible with the SoapUI tool, such as our custom JSON format. We have shown that this is possible through this test, thus showing that it is possible to feed incompatible data to a tool using our framework.

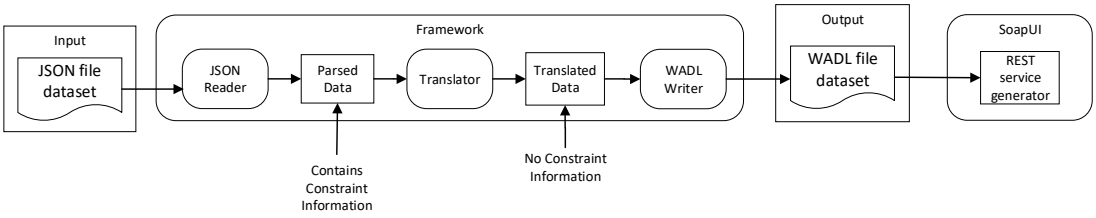

Figure 38: Feeding incompatible JSON data to the SoapUI REST generator

5.4 Realization of Quality Requirements

We laid out a series of non-functional requirements in Section 1.6 that we found by analyzing our motivation scenarios, presented in Section 1.5. In this section, we describe the tests we designed to verify that we have met these quality requirements with our solution, as well as their results.

5.4.1 Interoperability

We discussed in Section 1.6 that we required interoperability to achieve our goals. An important part of this is to achieve translation to allow solutions to use data in their respective formats. Specifically, we set out to support multiple Web service description models and to read and write from multiple sources and support multiple formats. We showed that we are capable of supporting WADL, WSDL, WSLA, our

custom JSON format, and our custom XML format, while representing both basic Web service description records and constrained Web service definition records, in Section 5.3.1, where we demonstrated our ability to translate datasets. Specifically, we demonstrated interoperability by supporting WADL, WSDL, WSLA, our custom JSON format, and our custom XML format independently, storing the data contained within in either basic Web service definition records or constrained Web service definition records. We also demonstrated the ability to translate from basic Web service definition records to constrained Web service definition records, and vice versa.

In Section 5.3.3, where we demonstrate our ability to combine datasets, we also showed that we can support WADL, WSDL, WSLA, and a PostgreSQL datbase, while representing both basic Web service description records and constrained Web service description records once again. In this case, we again demonstrate that we can translate from basic Web service definition records to constrained Web service definition records.

Additionally, to complete our fulfillment of interoperability, we set out to allow two previously incompatible solutions to communicate. In Section 5.2.6, we described how we allowed data formatted in our custom JSON format to be fed to the normally incompatible SoapUI [45] software for processing. Though the JSON file used was had-written, it could just have easily been produced by another program, thus showing that we can offer interoperability between two incompatible solutions. Thus, we have shown that our framework offers interoperability.

5.4.2 Adaptability

We discussed in Section 1.6 that adaptability was a requirement that would guide our efforts. We demonstrate that our framework can adapt to both composite and constrained Web service definition records. In Section 5.3.1 and Section 5.3.2, we demonstrated the use of our constrained Web service definition record hot spot. Though we have no tests that explicitly demonstrate our framework's ability to read and write composite Web service definition records, in Section 4.2, we presented our composite Web service description record hot spot, as we as our BPEL reader and

writer hot spots that allow us to read and write composite Web service definition records.

5.4.3 Extensibility

To meet our goals, we mention in Section 1.6 that we required our solution to be extensible. In Section 5.3.1, where we demonstrate our ability to translate datasets, we demonstrated that we can extend our framework with modules that perform the following tasks:

- Read WADL files.
- Read our custom JSON format.
- Read the custom XML format used by the WSC-gen Web service generator [14] as decribed in Section 4.3.
- Write WSDL files
- Write WSDL and WSLA files in conjunction.
- Extend the basic Web service definition record with constraints.
- Translate constrained Web service definition records to basic Web service definition records.
- Provide a command line interface to allow the user to interact with our framework and control the hot spots within.

In Section 5.3.2, where we demonstrate that our framework offers time scalability, we demonstrated that we can extend our framework with modules that perform the following tasks:

- Read WADL files.
- Read WADL files more efficiently.

- Write WSDL files.
- Write WSDL and WSLA files in conjunction.
- Extend the basic Web service definition record with constraints.
- Translate basic Web service definition records to constrained Web service definition records.
- Provide a command line interface to allow the user to interact with our framework and control the hot spots within.

In Section 5.3.3, where we demonstrate that our framework can combine datasets together, we demonstrated that we can extend our framework with modules that perform the following tasks:

- Read WADL files.
- Read WSDL files.
- Write WSDL files.
- Write WSDL and WSLA files in conjunction.
- Write constrained Web service definition records to a PostgreSQL database.
- Extend the basic Web service definition record with constraints.
- Translate basic Web service definition records to constrained Web service definition records.
- Provide a command line interface to allow the user to interact with our framework and control the hot spots within.

In Section 5.3.4, where we demonstrate that our framework offers space scalability, we demonstrated that we can extend our framework with modules that perform the following tasks:

- Read WADL files.
- Read WSDL files.
- Read constrained Web service definition records from a PostgreSQL database.
- Write constrained Web service definition records to a PostgreSQL database.
- Extend the basic Web service definition record with constraints.
- Translate basic Web service definition records to constrained Web service definition records.
- Provide a command line interface to allow the user to interact with our framework and control the hot spots within.

Section 5.3.5, where we demonstrate that we can add a new hot spot to our framework by directly integrating a new hot spot into our code. We explain that many of our hot spots, including the ones used to parse XML-based formats, such as WADL, are integrated into our framework in this way.

Section 5.3.6, where we demonstrate that we can add a new hot spot to our framework by importing a new hot spot as a plugin. We explain that some of our hot spots, notably our system interfaces, such as our command line sytem interface, are integrated into our framework in this way.

In Section 5.3.7, where we demonstrate the ability to communicate with our framework through HTTP, we demonstrated that we can extend our framework with modules that perform the following tasks:

- Read WSDL files.
- Write constrained Web service definition records in our custom XML format.
- Write constrained Web service definition records in our custom JSON format.
- Write basic Web service definition records in WADL.
- Write basic Web service definition records in WSDL.

- Write basic Web service definition records in serialized Java objects.
- Provide an HTTP interface to allow the user to interact with our framework and control the hot spots within.

In Section 5.3.8, where we demonstrate a local instance of our framework communicating with a public instance of our framework, we demonstrated that we can extend our framework with modules that perform the following tasks:

- Provide a server interface that allows the user to interact with our framework and control the hot spots within through a custom TCP protocol.
- Provide a client interface that allows the user to interact with our framework through a command line interface, as well as send commands to the previously mentioned server interface through TCP using our custom protocol.

In Section 5.3.9, where we demonstrate the ability to use our framework to take data that is incompatible with a tool and feed this data to the tool, we demonstrated that we can extend our framework with modules that perform the following tasks:

- Read files written in our custom JSON format.
- Write basic Web service definition records in WADL.
- Extend the basic Web service definition record with constraints.
- Translate constrained Web service definition records to basic Web service definition records.
- Provide a command line interface to allow the user to interact with our framework and control the hot spots within.

We also note that another student in our research group has leveraged and extended the functionality of our framework to produce algorithms related to service composition [57] as mentioned in Section 4.2 by using our framework as a library as described in Section 5.2.7.

We have thus shown that our framework provides extensibility, by demonstrating that we ourselves have extended our framework with all the previously mentioned modules.

5.4.4 Scalability

In Section 1.6 we explained that our solution would require both time and space scalability to be useful. We showed in Section 5.3.4 that we can improve a framework instance's space scalability by using a new hot spot that supports greater space scalability if required. In Section 5.3.3 we save data to both the file system and a PostgreSQL database to demonstrate that users can choose the hot spots that offer an appropriate amount of space scalability for their current needs. In Section 5.3.3, we mention that the data is saved across many WSDL and WSLA files on the file system. This is to account for the fact that one file containing all 72,967 Web service description records would be too large to feasibly load in memory and maintain efficiency. Our PostgreSQL database does not have this limitation as we can simply load a portion of the Web service description records from the database into memory.

We also showed in Section 5.3.2 that we can achieve greater time scalability by using a new hot spot that supports greater time scalability if required, replacing a less efficient hot spot with a more efficient one when greater efficiency is required. The bottleneck for space and time scalability is not the design of our framework, but rather within the hot spots that comprise a framework instance. This is due to the fact that our framework imposes no restrictions or limitations on how or where the data is stored or processed.

More generally, we have shown that the scalability of a specific instance of our framework is not limited by the frozen infrastructure of our framework, but rather by the hot spots included in this specific instance.

5.4.5 Usability

We mentioned in Section 1.6 that to achieve our goals, we must have a solution which demonstrates usability, or our solution could not be useful to help a researcher in their work. We have demonstrated in Section 5.3.8 that we can choose many different ways of extending and interfacing with our solution. One of these options include using our plugin system, allowing a researcher to extend our solution without even writing any code. We have also demonstrated in Section 5.3 that we can perform the tasks that we have listed in our motivation scenarios, such as translating and combining datasets, and using our framework to allow a tool to process data that would normally be incompatible. Thus, we have shown that a user can use our framework to correctly perform the tasks we claim our framework allows users to perform, while making this as easy as possible for the user.

5.5 Summary

In Section 5.2, we described the various methods that may be used to extend our solution. In Section 5.2.1, we described how one can manually instantiate a hot spot one has created oneself. In Section 5.2.2, we described how one can use our plugin manager to import a hot spot that one has created oneself. In Section 5.2.3, we described how one can manually instantiate a hot spot that was included in a library. In Section 5.2.4, we explained how one can import a hot spot included in a library using the plugin manager. In Section 5.2.5, we explained how one can create a program external to our framework that interfaces with our framework. In Section 5.2.6, we explained how we can allow a pre-existing solution that has no knowledge of our framework to communicate with our framework. In Section 5.2.7, we describeed how our framework can be used as a library to build a solution.

In Section 5.3, we described the tests we designed to verify that we can achieve our previously described motivation scenarios. We showed how we can translate a dataset in Section 5.3.1, how we achieved time scalability in Section 5.3.2, how we combined different datasets in Section 5.3.3, how we achieved space scalability in Section 5.3.4.

We then covered how we can extend our framework, the most basic method being creating a hot spot as described in Section 5.3.5, then we described how we allow researchers to import a plugin in Section 5.3.6 with less effort, how a researcher can use a public instance of our solution in Section 5.3.7 with little effort, and how a local instance can connect to a public instance in Section 5.3.8 to offer the advantages of all these approaches simultaneously. We then covered how we used translated data in a tool in Section 5.3.9.

Then, in Section 5.4, we described how our framework meets the quality requirements we have previously mentioned in Chapter 1. We verified that we met our goal of interoperability in Section 5.4.1, adaptability in Section 5.4.2, extensibility in Section 5.4.3, scalbility in Section 5.4.4, and usability in Section 5.4.5. In the next chapter, we examine the results of these tests to determine whether we were successful in achieving our goals.

Chapter 6

Conclusion and Future Work

This chapter briefly re-iterates the major points of this thesis and provides an interpretation of the results obtained in Chapter 5. We also detail the work we did not have the time to complete, and that we are currently still working on.

6.1 Overview

In Section 6.2, we list the final results of our work, along with our final thoughts. In Section 6.3, we list the limitations we have noticed in our solution. In Section 6.4, we lay out the future work we will be working on.

6.2 Final Results

We have shown, through the results of our tests, that our repository has allowed us to achieve the goals we have previously laid out in Section 1.7. This implies that our tool can be used in real-life research-based scenarios that enables researchers to effectively compare their novel solutions with other existing solutions, even though they might not be using the same underlying service description models. The existence of such a tool has the potential benefit of enabling better evidence-based research comparisons, which in turn will eventually enable the industry to adopt the best solutions for composite service generation and execution. We plan to make our tool available

publicly as a Git repository, possibly on GitHub, along with a large variety of diverse datasets that can be shared for the benefit of the research community.

Specifically, in Section 1.5, we described the motivation scenarios that would guide our work. We described in Section 1.5.1 a scenario where a researcher translates one dataset to a different format, Web service description model, or both. In Section 5.3.1 we showed that we can achieve this by translating two datasets. In Section 1.5.3 we described a scenario in which a user was not satisfied with the time scalability of a particular module, and simply replaced the module with a mode efficient one that performed the necessary computations in a more reasonable time. In Section 5.3.2, we showed that it is possible to do this with our framework by replacing a module with a much more efficient one to improve the runtime of a test we had previously described. In Section 1.5.2, we described a scenario in which a researcher needed to combine multiple datasets into one large dataset for their research. In Section 5.3.3 we showed that our framework can achieve this by combining two datasets and outputting the results to the file system. In Section 1.5.4, we explained that a researcher might find a particular module to offer insufficient space scalability and be incapable of handling a sufficient number of Web service description records. In Section 5.3.4, we showed that our framework allows a user to switch one module for another one that offers greater space scalability by creating reader and writer modules that can connect to a PostgreSQL database. In Section 1.5.6, we described how a researcher might want to create their own module connected to our framework while being as optimized as possible for this researcher's specific needs. In Section 5.3.5, we described how a researcher can achieve this by integrating a new hot spot directly into our code. In Section 1.5.7, we described how a researcher might want to leverage an existing solution that has already been created by another researcher. In Section 5.3.6, we described how a researcher can achieve this by importing plugin using our plugin manager. In Section 1.5.8, we mentioned that a researcher might not want to have to manage a local instance of our framework and simply want to leverage an existing instance of our framework that was available on a public server. In Section 5.3.7, we show that our framework can achieve this by creating a system interface that

provides an HTTP interface to our framework, and writing a python script that communicates with our framework using this interface to upload and download Web service description records in various formats. In Section 1.5.9, we mention that a researcher might want to combine aspects of all the previously mentioned ways of leveraging our framework in Section 5.3.5, Section 5.3.6, and Section 5.3.7. We show that our framework can achieve this in Section 5.3.8 by creating a pair of system interfaces, a client system interface and a server system interface, that communicate through TCP using a custom protocol. In Section 1.5.5, we describe how a researcher might want to take data translated by our framework and feed it to a tool for processing. In Section 5.3.9, we describe how we can use our framework to feed translated data to the SoapUI tool [45] to generate mock REST services.

In Section 1.6, we listed the non-functional requirements that we would need to meet to achieve the previously mentioned motivation scenarios listed in Section 1.5. In Section 5.4.1, we describe how we meet our requirement of interoperability. In Section 5.4.2, we explain how we achieve adaptability with our framework. In Section 5.4.3, we show how our framework achieves extensibility. In Section 5.4.4, we describe how we meet our requirement of scalability. In Section 5.4.5, we explain how our framework achieves usability.

We are confident that the inclusion of the crucial aspects that were missing from previous solutions as described in Section 1.4 will encourage users to adopt our solution. As our solution grows in popularity, more formats will be supported and increasingly varying datasets will be generated and used, which will encourage more users to adopt our solution and encourage more extensions and improvements. We hope to see our repository foster the development of new standard models, service description formats and Web service composition and execution algorithms that will eventually improve the research and development in the field.

6.3 Limitations

Even though our repository framework and the instance we have presented here have met all the requirements we had established, it currently has a few limitations.

Our current framework instance does not contain any hot spots that allow a user to execute or simulate the execution of Web services. This would provide a useful tool for researchers to develop and test their algorithms.

Our current framework instance does not contain any hot spots that give our framework access to a distributed storage system such as MongoDB and Hadoop. Using such solutions would virtually remove any practical limit to the size of the managed repository.

Our current hot spots only extracts part of the information contained in the WADL, WSDL, BPEL, PNML, and WSLA file formats, only extracting what is described in the models mentioned in Section 2.3.2. This leaves out, in some cases, most of the information that is contained within a file format, which could be stored in a Web service description record and used in designing algorithms.

Our framework instance does not currently have any reader hot spots that can handle files that are too large to completely fit in memory. This currently prevents us from having truly unlimited scalability.

We currently do not have a hot spot that automatically ensures that memory will not overflow such that no more Web service definition records can fit in. This somewhat limits the usefulness of our current hot spots, since users are currently responsible for ensuring that there is enough room left in memory to read in the Web service definition records contained within a file.

Our current framework instance contains no hot spots for reading or writing services described in natural language, such as services described in RESTful services. This means that a large number of Web service definition records are currently inaccessible to us.

Our current database reader hot spots offer no way of searching for services with queries more complex than giving the service's name. To truly exploit the power of databases, we need to support much more complex queries, such as searching by input, output, and constraint information.

Our current HTTP system interface hot spot only supports simple requests that save services sent to our server and provide all the services stored in our database when requested. To truly exploit our HTTP interface, we would need to support much more complex queries, such as viewing the available list of hot spots, and controlling of the available hot spots, including readers, writers, and translation layers.

Our plugin manager currently cannot access an online catalog to search for plugins and automatically download and install plugins as desired. Plugins must be installed by manually including a JRE file in our framework and sending the names of the classes contained within to the plugin manager to install them.

6.4 Future Work

The research presented in this thesis is only a small, though important, part of our general goal of coming up with a platform for experimentations on Web service composition and execution algorithms. Below, we list some additional features that we are working on that relate to the work presented here.

Other usage scenarios: There are more advanced and interesting scenarios that our solution could help realize. For example, one might want to compare two different algorithms (e.g., composite service execution algorithms) that use two different service models and thus use incompatible Web service description datasets. Our solution could be used to translate each dataset in to the other's format and run their own experiments using the other solution's dataset. This would enable the research community to compare solutions even though they are using different models and representation formats.

UDDI access: We consider offering our repository as a service, which can be accessed through UDDI for standardization. We could use the Apache jUDDI library for this purpose.

XML schema import: Instead of requiring users to write a new class to parse a new XML format, we plan to provide them with the possibility of using XML Schemas. This would add to the flexibility and ease of use of our framework. Using such a solution, updating the repository with new XML formats would no longer require recompilation.

Handling RESTful service descriptions: The ability to parse and write a format used by RESTful services would allow us to access services that are largely ignored in related works and allow us to access/manage a larger number and variety of services. For parsing RESTful services, we would need to create a module with natural language processing capabilities.

OpenISS: One of our other research endeavors involves using depth and regular cameras' cyber-physical IoT datasets with real physical constraints and QoS parameters from OpenISS-as-a-Service. We plan to develop framework hot spot components adapted to this purpose.

Complete WADL and BPEL support: Our WADL and BPEL support is currently limited to the model we have described in Section 2.3.2, since we do not currently have decorators that support all the information contained in either of these filetypes. Once that is supported, no information will be lost when reading and writing those formats.

Exporting to Lucid: We are working on using the Lucid dataflow language [58–61] to represent and execute composite services. In this case, composite service records can be thus translated into a Lucid program. Given that we have an execution engine for this language [62–65], we can execute the composite service represented. In cases where the service description data is artificial, our execution engine will be able to simulate the execution of the service in question. This will provide us with a composite service execution/simulation platform.

Handling very large files: We currently have no hot spots that are capable of

handling files that are too large to fit in memory. Some file formats can be read by reading blocks of the file from the harddrive one after the other, until the whole file has been read and all Web service definition records within have been parsed. It is possible to implement hot spots that provide this functionality, and we have plans to create some. For files that must be loaded completely into memory to be parsed, this solution would not work.

Dynamically Triggered Scalability: Though we have implemented hot spots that provide scalability by reading and writing to and from databases, we do not have a hot spot that dynamically writes all services to a database once there is too little space left in memory to read in more services. We intend to create some hot spots that implement this functionality. We are also considering the possibility of integrating this functionality directly into the frozen spots we have already created in a future version of our framework, to ensure that all hot spots automatically access this scalability.

Complex Database Queries: Though we have implemented a hot spot that can read from our PostgreSQL database, this does not exploit the full potential of our database. We will create new hot spots that allow users to search for Web service definition records in our database based on complex criteria, such as input, output, and constraint information.

Complex HTTP queries: We are considering extending the functionality of our current HTTP system interface to add support for more complex requests. For example, we will add the possibility of viewing the list of available hot spots, as well as controlling the provided hot spots. This will make out HTTP system interface much more useful.

Plugin Catalog: We currently do not have an online catalog to browse from that allows users to browse, download, and automatically install plugins to allow our plugin manager to add new hot spots quickly and easily to our framework. We plan to create one to be accessed through HTTP and allow users to search for plugins, browse available plugins, and select available plugins for download and automatic installation on a local instance of our framework.

Bibliography

[1] S. V. Hashemian and F. Mavaddat, "A graph-based approach to web services composition," in *The 2005 Symposium on Applications and the Internet*, pp. 183–189, Jan 2005.

[2] P. Wang, Z. Ding, C. Jiang, and M. Zhou, "Constraint-aware approach to web service composition," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 44, pp. 770–784, June 2014.

[3] A. Brogi and S. Corfini, "Behaviour-aware discovery of web service compositions," *International Journal of Web Services Research*, vol. 4, no. 3, p. 1, 2007.

[4] T. Laleh, J. Paquet, S. Mokhov, and Y. Yan, "Constraint verification failure recovery in web service composition," *Journal of Future Generation Computer Systems*, 2018. In press.

[5] A. Khodadadi, "Collection and classification of services and their context," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Sept. 2015. https://spectrum.library.concordia.ca/980615/.

[6] N. Steinmetz, "Web service location," Master's thesis, University of Innsbruck, Semantic Technology Institute (STI) Innsbruck, Aug. 2010. http://www.nathaliesteinmetz.net/master_thesis_nathalie_steinmetz.pdf. [accessed 20-January-2019].

[7] V. Kaewmarin, N. Arch-int, and S. Arch-int, "Semantic web service discovery and integration using service search crawler," in *International Conference on Computational Intelligence for Modelling Control & Automation*, pp. 884–888, 2008.

[8] E. Al-Masri and Q. H. Mahmoud, "WSCE: A crawler engine for large-scale discovery of web services," in *IEEE International Conference on Web Services (ICWS)*, (Salt Lake City, UT, USA), pp. 1104–1111, IEEE, 2007.

[9] T. Põld, *Heuristics for Crawling WSDL Descriptions of Web Service Interfaces - the Heritrix Case*. Bachelor's thesis, Institute of Computer Science, University of Tartu, Estonia, 2012.

[10] N. Steinmetz, H. Lausen, , and M. Brunner, "Web service search on large scale," in *Service-Oriented Computing* (L. Baresi, C.-H. Chi, and J. Suzuki, eds.), vol. 5900 of *Lecture Notes in Computer Science*, pp. 437–444, Springer Berlin Heidelberg, 2009.

[11] M. AbuJarour, F. Naumann, and M. Craculeac, "Collecting, annotating, and classifying public web services," in *On the Move to Meaningful Internet Systems: OTM 2010* (R. Meersman, T. Dillon, and P. Herrero, eds.), vol. 6426 of *Lecture Notes in Computer Science*, pp. 256–272, Springer Berlin Heidelberg, 2010.

[12] J. Wu, L. Chen, Y. Xie, and Z. Zheng, "Titan: a system for effective web service discovery," in *Proceedings of the 21st International Conference on World Wide Web (WWW '12 Companion)* (editor, ed.), ACM, 2012.

[13] WS-Challenge, "TestsetGenerator2009-1.3." [Online], 2009. https://code.google.com/p/wsc-pku-tcs/downloads/list. [accessed 20-January-2019].

[14] WS-Challenge and GIPSY R&D Group, "WSC-Gen." [online], 2017–2018. https://github.com/GIPSY-dev/WSC-Gen. [accessed 20-January-2019].

[15] ProgrammableWeb Team, "ProgrammableWeb." [Online], 2005–2018. https://www.programmableweb.com/. [accessed 20-January-2019].

[16] C. Peiris, "XMethods – web service listings." [Online], 2001–2018. http://www.chrispeiris.com/seminars/monash_uni/web_pages/XMethods%20-%20Web%20Service%20Listings.htm. [accessed 20-January-2019].

[17] MuleSoft, "A service repository for SOA governance." [online]. https://www.mulesoft.com/resources/esb/service-repository-registry. [accessed 20-January-2019].

[18] J. Ju. [online]. http://www.zjujason.com/data.html. [accessed 20-January-2019].

[19] J. Zhu, Z. Zheng, P. He, Y. Xiong, Y. Lu, and W. Team, "Towards open datasets and source code for web service recommendation." [online], 2009–2017. http://wsdream.github.io/. [accessed 20-January-2019].

[20] D. Dehua, "Deep web services crawler," *Dresden University of Technology*, 2010.

[21] T. Pold, "Heuristics for crawling wsdl descriptions of web service interfaces - the heritrix case," bachelor's thesis, University of Tartu, Faculty of Mathematics and Computer Science, 2012.

[22] T. Põld, *Heuristics for Crawling WSDL Descriptions of Web Service Interfaces - the Heritrix Case*. Bachelor's thesis, Institute of Computer Science, University of Tartu, Estonia, 2012.

[23] A. Keller and H. Ludwig, "The wsla framework: Specifying and monitoring service level agreements for web services," Tech. Rep. RC22456 (W0205-171), IBM Research Division, Yorktown Heights, NY, 2002. http://domino.watson.ibm.com/library/cyberdig.nsf/1e4115aea78b6e7c85256b360066f0d4/cdedb79080f59ee285256c5900654839.

[24] H. Ludwig, A. Keller, A. Dan, R. P. King, and R. Franck, *Web Service Level Agreement (WSLA) Language Specification*. IBM, Jan. 2003. http://www.research.ibm.com/people/a/akeller/Data/WSLASpecV1-20030128.pdf.

[25] P. Bianco, G. Lewis, and P. Merson, "Service level agreements in service-oriented architecture environments," Tech. Rep. CMU/SEI-2008-TN-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2008. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=8615.

[26] GIPSY R&D Group, "GIPSY R&D Group's simulation datasets." [Online], 2018. https://github.com/GIPSY-dev/datasets. [accessed 20-January-2019].

[27] TLDP, "Static libraries." [Online], 2013. http://tldp.org/HOWTO/Program-Library-HOWTO/static-libraries.html. [accessed 20-January-2019].

[28] IBM, "Shared libraries and shared memory." [Online], 2018. https://www.ibm.com/support/knowledgecenter/en/ssw_aix_72/com.ibm.aix.genprogc/shared_libs_mem.htm. [accessed 20-January-2019].

[29] Microsoft, "Dynamic-link libraries." [Online], 2018. https://docs.microsoft.com/en-us/windows/desktop/dlls/dynamic-link-libraries. [accessed 20-January-2019].

[30] W. Pree, "Meta Patterns - A Means For Capturing the Essentials of Reusable Object-Oriented Design," *ECOOP '94 Proceedings of the 8th European Conference on Object-Oriented Programming*, pp. 150–162, July 1994.

[31] A. Contributors, "AngularJS." [Online], 2018. https://github.com/angular/angular.js. [accessed 20-January-2019].

[32] Ember Contributors, "Ember." [Online], 2018. https://github.com/emberjs/ember.js. [accessed 20-January-2019].

[33] Meteor Contributors, "Meteor." [Online], 2018. https://github.com/meteor/meteor. [accessed 20-January-2019].

[34] Eclipse contributors *et al.*, "Eclipse Platform." eclipse.org, 2000–2014. http://www.eclipse.org, last viewed August 2014.

[35] Eclipse contributors *et al.*, "Equinox." [Online], 2000–2014. https://www.eclipse.org/equinox/. [accessed 20-January-2019].

[36] Google, "The Chromium Projects." [online], 2010. https://www.chromium.org/developers/design-documents/plugin-architecture. [accessed 20-January-2019].

[37] Atlassian, "Atlassian sdk developer." [Online], 2018. https://developer.atlassian.com/server/framework/atlassian-sdk/plugin-framework/. [accessed 20-January-2019].

[38] D. Riehle, "Framework design: A role modeling approach," *Swiss Federal Institute of Technology*, 2000. http://www.riehle.org/computer-science/research/dissertation/diss-a4.pdf.

[39] OSGi Alliance, "OSGi Alliance." [Online], 2018. https://www.osgi.org/. [accessed 20-January-2019].

[40] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, and The World Wide Web Consortium (W3C), "Web Services Description Language (WSDL) 1.1." [Online], Mar. 2001. http://www.w3.org/TR/wsdl. [accessed 20-January-2019].

[41] World Wide Web Consortium (W3C), "Web application description language." [online] http://www.w3.org/Submission/wadl/. [accessed 20-January-2019].

[42] OASIS Web Services Business Process Execution Language (WSBPEL) TC, "Web Services Business Process Execution Language version 2.0." [online], Oasis, Apr. 2007. OASIS Standard, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. [accessed 20-January-2019].

[43] PNML contributors, "Pnml.org." [online] http://www.pnml.org/. [accessed 20-January-2019].

[44] T. Laleh, J. Paquet, S. A. Mokhov, and Y. Yan, "Efficient constraint verification in service composition design and execution," in *On the Move to Meaningful*

Internet Systems: Proceedings of the OTM 2016 Conferences – Confederated International Conferences: CoopIS, C&TC, and ODBASE, Rhodes, Greece, October 24–28, 2016 (C. Debruyne, H. Panetto, R. Meersman, T. S. Dillon, E. Kühn, D. O'Sullivan, and C. A. Ardagna, eds.), vol. 10033 of *Lecture Notes in Computer Science*, pp. 445–455, 2016. Short paper.

[45] S. Software, "SoapUI." Published electronically, https://www.soapui.org/rest-testing/working-with-rest-services.html. [accessed 20-January-2019], 2018.

[46] A. S. Bilgin, "A daml-based repository for qos-aware semantic web service selection," *IEEE International Conference on Web Services*, vol. 2004, 2004. http://ieeexplore.ieee.org/abstract/document/1314759/.

[47] A. Barros and M. Dumas, "The rise of web service ecosystems," *IT Professional*, vol. 8, 2006. http://ieeexplore.ieee.org/abstract/document/1717340/.

[48] J. Yu, Q. Z. Sheng, J. Han, Y. Wu, and C. Liu, "A semantically enhanced service repository for user-centric service discovery and management," *Data & Knowledge Engineering*, vol. 72, 2012. https://www.sciencedirect.com/science/article/pii/S0169023X11001443.

[49] R. Hamadi and B. Benatallah, "A petri net-based model for web service composition," in *Proceedings of the 14th Australasian database conference-Volume 17*, pp. 191–200, Australian Computer Society, Inc., 2003.

[50] P. Sun, C. Jiang, and M. Zhou, "Interactive web service composition based on petri net," *Transactions of the Institute of Measurement and Control*, vol. 33, no. 1, pp. 116–132, 2011.

[51] R. Hamadi and B. Benatallah, "A petri net-based model for web service composition," in *Proceedings of the 14th Australasian database conference-Volume 17*, pp. 191–200, Australian Computer Society, Inc., 2003.

[52] H. N. Lakshmi and H. Mohanty, "Rdbms for service repository and composition," *Fourth International Conference on Advanced Computing*, vol. 2012, 2012. http://ieeexplore.ieee.org/abstract/document/6416810/.

[53] I. Mezgár and U. Rauschecker, "The challenge of networked enterprises for cloud computing interoperability," *Computers in Industry*, vol. 65, no. 4, pp. 657–674, 2014.

[54] T. Dillon, C. Wu, and E. Chang, "Cloud computing: issues and challenges," in *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, pp. 27–33, IEEE, 2010.

[55] Y. Demchenko, C. Ngo, M. Makkes, R. Stgrijkers, and C. de Laat, "Defining inter-cloud architecture for interoperability and integration," in *Proceedings of the The Third International Conference on Cloud Computing, GRIDs, and Virtualization (CLOUD COMPUTING'12)*, pp. 174–180, 2012.

[56] OpenESB Contributors, "Open Enterprise Service Bus (OpenESB)." [online], 2009–2018. http://www.open-esb.net/. [accessed 20-January-2019].

[57] J. Gupta, "Execution/simulation of context/constraint-aware composite services using GIPSY," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2019. unpublished.

[58] E. A. Ashcroft and W. W. Wadge, "Erratum: Lucid – a formal system for writing and proving programs," *SIAM J. Comput.*, vol. 6, no. 1, p. 200, 1977.

[59] S. C. Johnson, "A strategy for automatically generating programs in the Lucid programming language (NASA technical memorandum)," tech. rep., NASA, Scientific and Technical Information Office, 1987. ASIN: B000711R3Q.

[60] W. W. Wadge and E. A. Ashcroft, *Lucid, the Dataflow Programming Language*. London: Academic Press, 1985. ISBN: 978-0127296517.

[61] E. A. Ashcroft, A. A. Faustini, R. Jagannathan, and W. W. Wadge, *Multidimensional Programming*. London: Oxford University Press, Feb. 1995. ISBN: 978-0195075977.

[62] J. Paquet, "Distributed eductive execution of hybrid intensional programs," in *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09)*, pp. 218–224, IEEE Computer Society, July 2009.

[63] B. Han, S. A. Mokhov, and J. Paquet, "Advances in the design and implementation of a multi-tier architecture in the GIPSY environment with Java," in *Proceedings of the 8th IEEE/ACIS International Conference on Software Engineering Research, Management and Applications (SERA 2010)*, pp. 259–266, IEEE Computer Society, May 2010. Online at http://arxiv.org/abs/0906.4837.

[64] B. Han, "Towards a multi-tier runtime system for GIPSY," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2010.

[65] Y. Ji, "Scalability evaluation of the GIPSY runtime system," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Mar. 2011. http://spectrum.library.concordia.ca/7152/.